Type systems for programming languages

Didier Rémy

Academic year 2020-2021
Version of October 2, 2022

Contents

Chapter 4

Polymorphism and System F

4.1 Polymorphism

Polymorphism is the ability for a term to *simultaneously* admit several distinct types. Polymorphism is *indispensable* (Reynolds, 1974): if a list-sorting function is independent of the type of the elements, then it should be directly applicable to lists of integers, lists of booleans, *etc.*. In short, it should have polymorphic type:

$$\forall \alpha. (\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\ \alpha \to \mathsf{list}\ \alpha$$

which can then be *instantiated* to any of the monomorphic types:

$$(\mathsf{int} \to \mathsf{int} \to \mathsf{bool}) \to \mathsf{list\ int} \to \mathsf{list\ int} \qquad (\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}) \to \mathsf{list\ bool} \to \mathsf{list\ bool} \qquad \ldots$$

In the absence of polymorphism, the only ways of achieving this effect are either to manually duplicate the list-sorting function at every type (*no-no!*); or to use subtyping and claim that the function sorts lists of values of *any* type:

$$(\top \to \top \to \mathsf{bool}) \to \mathsf{list}\ \top \to \mathsf{list}\ \top$$

(The type $\top$ is the type of all values, and the supertype of all types.) This leads to *loss of information* and subsequently requires introducing an unsafe *downcast* operation. This was the approach followed in Java before generics were introduced in 1.5.

Moreover, polymorphism seems to come almost for free, as it is already implicitly present in simply-typed λ-calculus. Indeed, all types of the compose functions are

$$(\tau_1 \to \tau_2) \to (\tau_0 \to \tau_1) \to \tau_0 \to \tau_2$$

among which is

$$(\alpha_1 \to \alpha_2) \to (\alpha_0 \to \alpha_1) \to \alpha_0 \to \alpha_2$$

which is principal, as all other types can be recovered by instantiation of the variables. By saying that this term admits the polymorphic type

$$\forall \alpha_1 \alpha_2. (\alpha_1 \to \alpha_2) \to (\alpha_0 \to \alpha_1) \to \alpha_0 \to \alpha_2$$

we make polymorphism *internal* to the type system.

Polymorphism is a step on the road towards *type abstraction.* Intuitively, if a function that sorts a list has polymorphic type

$$\forall \alpha. (\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\ \alpha \to \mathsf{list}\ \alpha$$

then it *knows nothing* about α—it is *parametric* in α—so it must manipulate the list elements *abstractly:* it can copy them around, pass them as arguments to the comparison function, but it cannot directly inspect their structure. In short, within the code of the list sorting function, the variable α is an *abstract type.*

Parametricity In the presence of polymorphism (and in the absence of effects), a type can reveal a lot of information about the terms that inhabit it. For instance, the polymorphic type $\forall \alpha. \alpha \to \alpha$ has only one inhabitant, namely the identity. Similarly, the type of the list sorting function

$$\forall \alpha. (\alpha \to \alpha \to \mathsf{bool}) \to \mathsf{list}\ \alpha \to \mathsf{list}\ \alpha$$

reveals a *"free theorem"* about its behavior! Basically, sorting commutes with (map f), provided f is order preserving. Note that there are many inhabitants of this type (e.g. a function that sorts in reverse order, or a function that removes duplicates) but they all satisfy this free theorem. This phenomenon was studied by Reynolds 1983 and by Wadler 1989; 2007, among others. An account based on an operational semantics is offered by Pitts 2000.

***Ad hoc* versus *parametric* polymorphism** Let us begin a short digression. The term "polymorphism" dates back to a 1967 paper by Strachey (2000), where *ad hoc polymorphism* and *parametric polymorphism* were distinguished. There are two different (and sometimes incompatible) ways of defining this distinction:

- With parametric polymorphism, a term can admit several types, all of which are *instances* of a common polymorphic type: $\mathsf{int} \to \mathsf{int}$, $\mathsf{bool} \to \mathsf{bool}$, ... and $\forall \alpha. \alpha \to \alpha$.

  With ad hoc polymorphism, a term can admit a collection of *unrelated* types: $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$, $\mathsf{float} \to \mathsf{float} \to \mathsf{float}$, ... but *not* $\forall \alpha. \alpha \to \alpha \to \alpha$.

- With parametric polymorphism, *untyped programs have a well-defined semantics.* (Think of the identity function.) Types are used only to rule out unsafe programs.

  With ad hoc polymorphism, untyped programs do not have a semantics: *the meaning of a term can depend upon its type* (e.g. $2+2$), or, even worse, *upon its type derivation* (e.g. $\lambda x. \mathit{show}\ (\mathit{read}\ x)$).

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \textsc{Var} \qquad \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, M : \tau_1 \to \tau_2}\ \textsc{Abs} \qquad \frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\ M_2 : \tau_2}\ \textsc{App}$$

$$\frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\tau}\ \textsc{Tabs} \qquad \frac{\Gamma \vdash M : \forall\alpha.\tau}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}\ \textsc{Tapp}$$

Figure 4.1: Typing rules for System F.

By the first definition, Haskell's *type classes* (Hudak et al., 2007) are a form of (bounded) parametric polymorphism: terms have *principal (qualified) type schemes*, such as:

$$\forall\alpha.\, \mathsf{Num}\ \alpha \Rightarrow \alpha \to \alpha \to \alpha$$

Yet, by the second definition, type classes are a form of ad hoc polymorphism: untyped programs do not have a semantics. This ends the digression.

4.2 Polymorphic λ-calculus

The System F, (also known as: the polymorphic λ-calculus; the second-order λ-calculus; F_2) was independently defined by Girard (1972) and Reynolds (1974).

4.2.1 Types and typing rules

Types of the simply-typed λ-calculus are extended with polymorphic types:

$$\tau ::= \alpha \mid \tau \Rightarrow \tau \mid \forall\alpha.\tau$$

How are the syntax and semantics of terms extended? There are several variants, depending on whether one adopts an *implicitly-typed* or *explicitly-typed* presentation of terms and a *type-passing* or a *type-erasing* semantics.

In the explicitly-typed variant (Reynolds, 1974), there are term-level constructs for introducing and eliminating the universal quantifier (we recall the previous rules of simply-typed λ-calculus in gray):

$$M ::= x \mid \lambda x{:}\tau.\, M \mid M\ M \mid \Lambda\alpha.M \mid M\ \tau$$

We write F for the set of explicitly-typed terms.

Type variables are explicitly bound and appear in type environments:

$$\Gamma ::= \varnothing \mid \Gamma, x : \tau \mid \Gamma, \alpha$$

We extend our previous convention to form environments: Γ, α extends Γ with a new variable α, provided $\alpha \# \Gamma$, *i.e.* α is neither in the domain nor in the image of Γ. We also require that environments be closed with respect to type variables. That is, we require $\mathsf{ftv}(T) \subseteq \mathsf{dom}(\Gamma)$ to form $\Gamma, x : \tau$. This additional requirement is a matter of convenience. It allows fewer judgments, since judgments with open contexts are not allowed. However, open contexts can always be closed by adding a prefix composed of a sequence of its free type variables. Hence, a loose definition of contexts (without this requirement) can also be used, and the differences would be insignificant.

Well-formedness of environments and types may be defined (recursively) by inference rules (Rule WfEnvVar depends on well-formedness of types while Rule WfTypeVar depends on well-formedness of environments):

$$\textsc{WfEnvEmpty}\quad \vdash \varnothing \qquad \textsc{WfEnvTvar}\quad \frac{\vdash \Gamma \qquad \alpha \notin \mathsf{dom}(\Gamma)}{\vdash \Gamma, \alpha} \qquad \textsc{WfEnvVar}\quad \frac{\Gamma \vdash \tau \qquad x \notin \mathsf{dom}(\Gamma)}{\vdash \Gamma, x : \tau}$$

$$\textsc{WfTypeVar}\quad \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{WfTypeArrow}\quad \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \qquad \textsc{WfTypeForall}\quad \frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha.\, \tau}$$

Notice the absence of the premises $\vdash \Gamma$ in Rule WfEnvVar, since well-formedness of Γ is recursively implied by the well-formedness of τ in Γ. Similarly, the well-formedness of Γ is required in Rule WfTypeVar but not in rules WfTypeArrow and WfTypeForall.

There is a choice whether well-formedness of environments should be made explicit or left implicit in typing rules.

Explicit well-formedness amounts to adding well-formedness premises to every rule where the environment or some type that appears in the conclusion did not appear in any premise. Namely:

$$\textsc{Var}\quad \frac{x : \tau \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Tapp}\quad \frac{\Gamma \vdash M : \forall \alpha.\tau \qquad \Gamma \vdash \tau'}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}$$

Explicit well-formedness is more precise and better suited for mechanized proofs. It is also recommended for (more) complicated type systems. However, it is a bit verbose and distracting for System F. The two styles are really equivalent. Formally, we choose to leave well-formedness implicit. However, for documentation purposes, we will indicate the well-formedness premises in the definition of typing rules.

4.2.2 Semantics

We need the following reduction for type abstraction:

$$(\Lambda \alpha.M)\ \tau \longrightarrow [\alpha \mapsto \tau]M \tag{ι}$$

Then, there is a choice regarding whether type abstraction should stop the evaluation, or let reduction proceed.

Type-passing semantics In most presentations of System F, type abstraction blocks the evaluation and is defined as follows:

$$E ::= [\,]\ M \mid V\ [\,] \mid [\,]\ \tau \qquad\qquad V ::= \lambda x\!:\!\tau.\, M \mid \Lambda\alpha.M$$

This is a *type-passing* semantics. Indeed, $\Lambda\alpha.((\lambda y : \alpha.\, y)\ V)$ is a value while its type erasure is $(\lambda y.\, y)\ \lceil V \rceil$ is not—and can be further reduced.

The type-passing semantics is perhaps more natural in a language with a call-by-value semantics since type abstraction stops evaluation exactly as value abstraction.

However, it does not fit our view that *the untyped semantics should pre-exist* and that a type system is only a predicate that selects a subset of the well-behaved terms, since type abstraction alters the semantics.

In particular, it introduces a discontinuity between monomorphic and polymorphic types. Assume for example that f is list flattening of type $\forall\alpha.\, \mathsf{list}\ (\mathsf{list}\ \alpha) \to \mathsf{list}\ \alpha$ and $\circ$ is the composition function $\Lambda\alpha_1.\Lambda\alpha_0.\Lambda\alpha_2.\lambda f : \alpha_0 \to \alpha_2.\, \lambda g : \alpha_1 \to \alpha_0.\, \lambda x : \alpha_1.\, f\ g\ x$; then, the monomorphic function $(f\ \mathsf{int})\ (\circ\ \mathsf{int}\ (\mathsf{list}\ \mathsf{int})\ (\mathsf{list}\ (\mathsf{list}\ \mathsf{int})))\ (f\ (\mathsf{list}\ \mathsf{int}))$ reduces to $\lambda x\!:\!\mathsf{int}.\, f\ \mathsf{int}\ (f\ (\mathsf{list}\ \mathsf{int})\ x)$, while its more general polymorphic version

$$\Lambda\alpha.(f\ \alpha)\ (\circ\ \alpha\ (\mathsf{list}\ (\mathsf{list}\ \alpha))\ (\mathsf{list}\ (\mathsf{list}\ \alpha)))\ (f\ (\mathsf{list}\ \alpha))$$

is irreducible. This discontinuity is disturbing especially in an implicitly-typed language such as ML, where type inference infers the most general version, which behaves less efficiently than its less general monomorphic variant.

Furthermore, since the type-passing semantics requires both values and types to exist at runtime, it can lead to a *duplication of machinery*. Compare type-preserving closure conversion in type-passing (Minamide et al., 1996) and in type-erasing (Morrisett et al., 1999) styles.

Type-erasing semantics To recover a type-erasing semantics (also called an *untyped semantics*), we need to allow evaluation under type abstraction:

$$E ::= [\,]\ M \mid V\ [\,] \mid [\,]\ \tau \mid \Lambda\alpha.[\,] \qquad\qquad V ::= \lambda x\!:\!\tau.\, M \mid \Lambda\alpha.V$$

Accordingly, we only need a weaker version of ι-reduction:

$$(\Lambda\alpha.V)\ \tau \longrightarrow [\alpha \mapsto \tau]V \qquad (\iota_v)$$

We now have:

$$\Lambda\alpha.(\lambda y : \alpha.\, y)\ V \longrightarrow \Lambda\alpha.V$$

We will show below that this defines a type-erasing semantics, indeed.

As an apparent drawback, the type-erasing semantics does not allow a *typecase*; however, typecase can be simulated by viewing runtime *type descriptions* as *values* (Crary et al., 2002).

On the opposite the *type-erasing* semantics, has several advantages: it does not alter the semantics of untyped terms; it coincides with the semantics of ML—and, more generally, with the semantics of most programming languages. It also exhibits difficulties when adding side effects while the type-passing semantics keeps them hidden.

For all these reasons, we prefer the type-erasing semantics, which we chose in the rest of this course. Notice that we allow evaluation under a type abstraction as a consequence of choosing a type-erasing semantics—and not the converse.

The two views may be reconciled by restricting type abstraction to value-forms (which include values and variables), that is, by only allowing value-forms $\Lambda\alpha.M$ when M is itself a value-form. Under this restriction, the type-passing and type-erasing semantics coincide. Indeed, closed type abstractions are then always type abstraction of values, and evaluation under type abstraction even if allowed may never be used. We will choose this restriction as a way to preserve type soundness when adding side effects to the language.

Implicitly-typed *v.s.* explicitly-typed variants We presented the *explicitly-typed* variant of System F. This is simpler for the meta-theoretical study while the implicitly typed version, and in particular its interesting ML subset, may be more convenient to use in practice. Fortunately, most meta-theoretical properties of the explicitly-typed version can then be transferred to the implicitly-typed version—so that proofs do not have to be redone in a different setting when putting theory into practice!

4.2.3 Extended System F with datatypes

System F is quite expressive: it enables the encoding of data structures. For instance, the Church encoding of pairs in the untyped λ-calculus is actually well-typed in System F:

$$\begin{array}{rcl}
Pair & \triangleq & \Lambda\alpha_1.\Lambda\alpha_2.\lambda x_1{:}\alpha_1.\,\lambda x_2{:}\alpha_2.\,\Lambda\beta.\lambda y{:}\alpha_1 \to \alpha_2 \to \beta.\, y\; x_1\; x_2 \\
proj_i & \triangleq & \Lambda\alpha_1.\Lambda\alpha_2.\lambda y{:}\forall\beta.\,(\alpha_1 \to \alpha_2 \to \beta) \to \beta.\, y\; \alpha_i\; (\lambda x_1{:}\alpha_1.\,\lambda x_2{:}\alpha_2.\, x_i) \\
\lceil Pair \rceil & \triangleq & \lambda x_1.\,\lambda x_2.\,\lambda y.\, y\; x_1\; x_2 \\
\lceil proj_i \rceil & \triangleq & \lambda y.\, y\; (\lambda x_1.\,\lambda x_2.\, x_i)
\end{array}$$

Notice the use of first-class polymorphism in the definition of $proj_i$. This is general in the encoding of datatypes.

Natural numbers, List, *etc.* can also be encoded.

Unit, Pairs, Sums, *etc.* can also be added to System F as primitives. We can then proceed as for simply-typed λ-calculus. However, we may also take advantage of the expressive type system of System F to deal with such extensions in a more elegant way: thanks to polymorphism, we need not add new typing rules for each extension. We may instead add one typing rule for constants and parametrize the definition by an initial typing environment Δ for constants. This allows sharing the meta-theoretical developments between the different extensions.

Adding primitive pairs Let us first illustrate datatypes on an example, adding primitive pairs to System F. We will then generalize the presentation to parametrize the extension as suggested above.

We introduce a new type constructor $(\cdot \times \cdot)$ of arity 2 to classify pairs:

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall \alpha.\, \tau \mid \tau \times \tau$$

Expressions are extended with a constructor $(\cdot, \cdot)$ and two destructors proj_1 and proj_2 with the respective signatures:

$$\begin{array}{rl} \mathit{Pair}: & \forall \alpha_1.\, \forall \alpha_2.\, \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2 \\ \mathit{proj}_i: & \forall \alpha_1.\, \forall \alpha_2.\, \alpha_1 \times \alpha_2 \to \alpha_i \end{array}$$

that forms the initial typing environment Δ. We need not add any new typing rule, but instead type programs in the initial environment Δ.

This allows for the formation of partial applications of constructors and destructors. Hence, values are extended as follows:

$$\begin{array}{rcl} V & ::= & \ldots \mid \mathit{Pair} \mid \mathit{Pair}\ \tau \mid \mathit{Pair}\ \tau\ \tau \mid \mathit{Pair}\ \tau\ \tau\ V \mid \mathit{Pair}\ \tau\ \tau\ V\ V \\ & & \mid \mathit{proj}_i \mid \mathit{proj}_i\ \tau \mid \mathit{proj}_i\ \tau\ \tau \end{array}$$

We add the two following reduction rules:

$$\mathit{proj}_i\ \tau_1\ \tau_2\ (\mathit{Pair}\ \tau_1'\ \tau_2'\ V_1\ V_2) \longrightarrow V_i \qquad (\delta_{pair})$$

Notice that, for well-typed programs, τ_i and τ_i' will always be equal, but the reduction will not check this at runtime. This could be enforced by replacing δ with the following rule:

$$\mathit{proj}_i\ \tau_1\ \tau_2\ (\mathit{Pair}\ \tau_1\ \tau_2\ V_1\ V_2) \longrightarrow V_i \qquad (\delta_{pair}')$$

The two semantics coincide on well-typed terms, but differ on ill-typed terms where δ_{pair}' may block when rule δ_{pair} would progress, ignoring type errors. Interestingly, using δ_{pair}' simplifies the proof obligation in subject reduction but introduces a more stronger proof obligation in progress.

Notice that since pairs are defined by applying the pair constructor to two arguments, the programmer must first specify the types of the components although those could be uniquely determined from the arguments of the pair. Even though this is a bit more verbose that strictly necessary, it should not be considered as a problem in an explicitly-typed presentation, as removing redundant type annotations is the task of type reconstruction.

A general approach Adding other datatypes such as booleans, integers, strings, lists, trees, *etc.* and operations on them can be done similarly. However, all these extensions are quite similar. Hence, we propose a general approach for adding constants to System F, which can then be instantiated independently—or simultaneously—to each of the previous cases: provided the dynamic semantics of constraints agree with their static semantics (some requirements must be satisfied in order to instantiate the general approach), the soundness of the extension then automatically follows.

We assume given a collection of constants, written with letter c, each of which given with a fix arity written $\mathsf{arity}\,(c)$. Constants must actually be partitioned into constructors (written C) and destructors (written d); moreover, we disallow nullary destructors[1].

Expressions are extended with constant expressions.

$$M ::= x \mid \lambda x : \tau .\, M \mid M\ M \mid \Lambda \alpha . M \mid M\ \tau \mid c$$

The difference between constructors and destructors lies in the fact that full application of constructors are values while full applications of destructors are not—they must be reduced. Partial applications of constants are always values. Hence, the following definition of values:

$$V ::= \lambda x : \tau .\, M \mid \Lambda \alpha . V \mid C\, \tau_1 \ldots\ \tau_i\, V_1 \ldots\ V_n \mid d\, \tau_1 \ldots\ \tau_j\, V_1 \ldots\ V_k$$

where n is less or equal to the arity of C and k is strictly less than the arity of d. The semantics of constants is given by providing, for each destructor d a relation δ_d defined by a set of δ-rules of the form:

$$d\, \tau_1 \ldots\ \tau_j\, V_1 \ldots\ V_k \longrightarrow M \qquad (\delta_d)$$

We assume given a collection of type constructors G, with their arity, written $\mathsf{arity}\,(\mathsf{G})$. Types are extended as follows.

$$\tau ::= \ldots \mid \mathsf{G}\, \tau_1 \ldots \tau_n$$

We assume that types respect the arities of type constructors, *i.e.* n is equal to $\mathsf{arity}\,(\mathsf{G})$ in the expressions $\mathsf{G}\, \tau_1 \ldots \tau_n$.

The typing of constants is given by the initial typing environment Δ. which binds each constant c of arity n to a type of the form $\forall \alpha_1. \ldots \forall \alpha_j. \tau_1 \to \ldots \tau_n \to \tau$. When c is a constructor C, we require that the top most type constructor of τ not be an arrow, but some type constructor G. We then say that C is a G-constructor. We require that Δ be well-formed (in the empty environment, hence closed). Constants are typed as variables, except that their types are looked up in Δ:

$$\textsc{Cst}\qquad \frac{c : \tau \in \Delta \qquad \vdash \Gamma}{\Gamma \vdash c : \tau}$$

Taking typing constraints into account, we may give a more restrictive characterization of well-typed values: in the presentation above i is at most the number of quantified variables in the type scheme of the constructor, and whenever n is non zero, i is equal to this number. And

[1]Nullary polymorphic destructors introduce pathological cases to maintain the semantics type-erasing—for little benefit in return.

similarly for destructors. For instance, if C is a constructor (respectively, d is a destructor) of arity q and of type $\forall \alpha_1 \dots \alpha_p . \tau'_1 \to \dots \tau'_q \to \tau$, then values will contain:

$$C \mid C\,\tau_1 \mid \dots\ C\,\tau_1\ \dots \tau_p \mid C\,\tau_1\ \dots \tau_p\,V_1 \mid \dots\ C\,\tau_1\ \dots \tau_p\,V_1\ \dots V_q$$

and

$$c \mid c\,\tau_1 \mid \dots\ c\,\tau_1\ \dots \tau_p \mid c\,\tau_1\ \dots \tau_p\,V_1 \mid \dots\ c\,\tau_1\ \dots \tau_p\,V_1\ \dots V_{q-1}$$

Of course, we need assumptions to relate typing and reduction of constants.

Definition 1 *δ-reduction is* sound *if it preserves typings and ensures progress for primitives. That is*

- *If $\vec{\alpha} \vdash M_1 : \tau$ and $M_1 \longrightarrow_\delta M_2$ then $\vec{\alpha} \vdash M_2 : \tau$.*
- *If $\vec{\alpha} \vdash M_1 : \tau$ and M_1 is of the form $d\,\tau_1\ \dots \tau_k\,V_1\ \dots V_n$ where $n = \mathsf{arity}\ (d)$, then there exists M_2 such that $M_1 \longrightarrow_\delta M_2$.*

Intuitively, progress for constants means that the domain of destructors is at least as large as specified by their type in Δ.

We will show below that soundness of δ-rules is sufficient to ensure soundness of the extension.

For example, to add a unit constant, we only introduce a type constant unit and a constructor $()$ of arity 0 of type unit. As no primitive is added, δ-reduction is obviously sound. Hence, the extension of System F with unit is sound.

Exercise 24 (Pairs as constants) *Reformulate the extension of System F with pairs as constants. Check soundness of the δ-rules.* *(Solution p. 84)* □

Exercise 25 (Conditional) *Give a presentation of boolean with a conditional as constants. Is this sound? Isn't there something wrong? Would you know how to fix it?*
(Solution p. 84) □

Exercise 26 (List) *1) Formulate the extension of System F with lists as constants. 2) Check that this extension is sound.* *(Solution p. 84)* □

Extending System F with a fixpoint The call-by-value fixpoint combinator Z (see §2) is not typable in System F—indeed this would allow program to loop while all programs terminate in System F.

However, we may introduce a fixpoint as a binary primitive with the following typing assumption:

$$\mathsf{fix} : \forall \alpha. \forall \beta. ((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta \qquad \in \Delta$$

and the reduction rule:

$$\mathsf{fix}\ \tau_1\ \tau_2\ V_1\ V_2 \longrightarrow V_1\ (\mathsf{fix}\ \tau_1\ \tau_2\ V_1)\ V_2 \qquad (\delta_{\mathsf{fix}})$$

It is straightforward to check the soundness of this extension: Progress is by construction, since fix does not destruct values. As for subject reduction, assume $\Gamma \vdash \mathsf{fix}\, \tau_1\, \tau_2\, V_1\, V_2 : \tau$. By inversion of typing rules, τ must be equal to τ_2, V_1 and V_2 must be of respective types $(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$ and τ_1 in the typing context Γ. We may then easily build a derivation of the judgment $\Gamma \vdash V_1\, (\mathsf{fix}\, \tau_1\, \tau_2\, V_1)\, V_2 : \tau$.

Exercise 27 (Recursion with datatypes) *In ML a one-constructor datatype can be used to emulate recursive types, namely a type* Any *such that a value of type* $\mathsf{any} \to \mathsf{any}$ *can be converted to a value of type* any*, and conversely. Give the definition in ML. Describe the extension as the addition of new constants. Verify the soundness of δ-rules.*

Use this extension to define a call-by-value fixpoint operator of type

$$((\mathsf{any} \to \mathsf{any}) \to \mathsf{any} \to \mathsf{any}) \to \mathsf{any} \to \mathsf{any}$$

in ML without using `let rec` *or implicit recursive types (the −rectypes option). (See Exercise 7 for a definition of the fix-point in the λ-calculus or in ML with recursive types.)*

(Solution p. 85) □

4.3 Type soundness

We prove type soundness for System F with constants, assuming the soundness of δ-reduction.

The structure of the proof is similar to the case of simply-typed λ-calculus and follows from subject reduction and progress. Subject reduction uses the following auxiliary lemmas: inversion of typing rules (Lemma 13), permutation (Lemma 14), weakening (Lemma 15), expression substitution (Lemma 16), type substitution (Lemma 17), and compositionality of typing (Lemma 18).

Lemma 13 (Inversion of typing rules) *Assume* $\Gamma \vdash M : \tau$*.*

- *If M is a variable x, then $x \in \mathsf{dom}(\Gamma)$ and $\Gamma(x) = \tau$.*
- *If M is $\lambda x{:}\tau_0. M_1$, then τ is of the form $\tau_0 \to \tau_1$ and $\Gamma, x : \tau_0 \vdash M_1 : \tau_1$.*
- *If M is $M_1\, M_2$ then $\Gamma \vdash M_1 : \tau_2 \to \tau$ and $\Gamma \vdash M_2 : \tau_2$ for some type τ_2.*
- *If M is a constant c, then $c \in \mathsf{dom}(\Delta)$ and $\Delta(x) = \tau$.*
- *If M is $M_1\, \tau_2$ then τ is of the form $[\alpha \mapsto \tau_2]\tau_1$ and $\Gamma \vdash M_1 : \forall \alpha. \tau_1$.*
- *If M is $\Lambda\alpha.M_1$, then τ is of the form $\forall \alpha. \tau_1$ and $\Gamma, \alpha \vdash M_1 : \tau_1$.*

Lemma 14 (Permutation) *If Γ and Γ' are two well-formed permutations, then $\Gamma \vdash M : \tau$ iff $\Gamma \vdash M : \tau$.*

Proof: Formally, the proof is by induction on M. The key is the observation that when Γ and Γ' are both well-formed and permutations of one another, they are equivalent as partial functions, *i.e.* they give the same bindings and can be extended in the same manner.

Lemma 15 (Weakening) *If* $\Gamma \vdash M : \tau$ *and* $\vdash \Gamma, \Gamma'$, *then* $\Gamma, \Gamma' \vdash M : \tau$.

Proof: It suffices to prove the lemma when Γ' is either $x : \tau'$ or α, since the general case follows by induction on the length of Γ'. We may prove both simultaneously, by induction on M. The proof is similar to the one for simply-typed λ-calculus—we just have more cases. Cases for value and type abstraction appeal to the permutation lemma. More precisely:

Case M is y: By inversion of typing, the judgment must be derived with rule Var, hence $y : \tau$ is in Γ and a fortiori $y : \tau$ is in Γ, Γ'. We may thus conclude by rule Var.

Case M is c: By inversion of typing, the judgment must be derived with rule Cst, hence we have $y : \tau$ is in Δ and we may conclude with rule Cst.

Case M is $\lambda y \colon \tau_1 . M_2$: *W.l.o.g.* we may choose y disjoint from Γ and Γ' (**1**). By inversion of typing, the judgment must be derived with rule Abs, hence $\Gamma, y : \tau_1 \vdash M_1 : \tau_2$ where τ is $\tau_1 \to \tau_2$. Since $\Gamma, y : \tau$ is well-formed, by (1), both $\Gamma, y : \tau_1, \Gamma'$ and $\Gamma, \Gamma', y : \tau_1$ are well-formed (**2**). By induction hypothesis, we have $\Gamma, x : \tau_1, \Gamma' \vdash M_1 : \tau_2$. Using the permutation lemma and (2), we have $\Gamma, \Gamma', x : \tau_1 \vdash M_1 : \tau_2$. We conclude with rule Abs.

Case M is $\Lambda\beta . M_1$: *W.l.o.g*, we may choose β disjoint from Γ and Γ' (**3**). By inversion of typing, the judgment must be derived with rule TAbs, hence $\Gamma, \beta \vdash M_1 : \tau_1$ with $\forall \beta . \tau_1$ equal to τ. Since Γ, β is well-formed, by (3), both Γ, β, Γ' and Γ, Γ', β are well-formed (**4**). By induction hypothesis, we have $\Gamma, \beta, \Gamma' \vdash M_1 : \tau_1$. We use the permutation lemma to obtain $\Gamma, \Gamma', \alpha \vdash M_1 : \tau_1$ and conclude with Rule Tabs.

Case M is $M_1\ M_2$ or $M_1\ \tau_1$: By inversion of typing, induction hypothesis applied to the premises, and App or TApp to conclude.

Lemma 16 (Expression substitution, strengthened)
If $\Gamma, x : \tau_0, \Gamma' \vdash M : \tau$ *and* $\Gamma \vdash M_0 : \tau_0$ *then* $\Gamma, \Gamma' \vdash [x \mapsto M_0]M : \tau$.

We have strengthened the lemma with an arbitrary context Γ' as for the simply-typed λ-calculus. We have also generalized the lemma with an arbitrary context Γ on the left and an arbitrary expression M, as this does not complicate the proof (and the stronger result will be used later). The proof is similar to the one for the simply-typed λ-calculus, with just a few more cases. (Details of the proof p. 85)

Exercise 28 *Write the details of the proof.* □

Lemma 17 (Type substitution, strengthened)
If $\Gamma, \alpha, \Gamma' \vdash M : \tau$ *and* $\Gamma \vdash \tau_0$ *then* $\Gamma, \theta\Gamma' \vdash \theta M : \theta\tau$ *where* θ *is* $[\alpha \mapsto \tau_0]$.

As for expression substitution, we have strengthened the lemma and generalized it using an arbitrary environment instead of the empty environment, as it does not complicate the proof, but yields a stronger result. This lemma resembles the one for expression substitutions. However, the substitution must also apply to the environment Γ' and the result type τ since α may appear free in them.

The proof is by induction on M. The interesting cases are for type and value abstraction, which required the strengthened version with an arbitrary typing context Γ' on the right. Then, the proof is straightforward. (Details of the proof p. 86)

Exercise 29 *Write the details of the proof.* □

Lemma 18 (Compositionality) *If* $\Gamma \vdash E[M] : \tau$*, then there exists a sequence of type variables* $\vec{\alpha}$ *and* τ' *such that* $\Gamma, \vec{\alpha} \vdash M : \tau'$ *and all* M' *verifying* $\Gamma, \vec{\alpha} \vdash M' : \tau'$ *also verify* $\Gamma \vdash E[M'] : \tau$.

Proof: The proof is by case on E. Each case is easy. The main difference with the simply-typed λ-calculus is that the case for type abstraction $\Lambda\alpha.E_0$ requires to extend the environment with type variables.

Notice that M' is typechecked in the context Γ extended with $\vec{\alpha}$, since the hole in the context E may be under type abstractions. We use the notation $\vec{\alpha}$ for a (possibly empty) sequence of type variables.

Theorem 9 (Subject Reduction) *Reduction preserves typings.*
If $\Gamma \vdash M : \tau$ *and* $M \longrightarrow M'$ *then* $\Gamma \vdash M' : \tau$.

The proof is by induction over the derivation of $M \longrightarrow M'$. Using the previous lemmas and the subject-reduction assumption for δ-reduction, the proof is straightforward.

Proof: By induction over the derivation of $M \longrightarrow M'$, then by inversion of the typing derivation of $\Gamma \vdash M : \tau$ (**1**).

Case $(\lambda x{:}\tau_1.\, M_1)\ V \longrightarrow [x \mapsto V]M_1$: By inversion, the typing derivation of (1) is of form:

$$\dfrac{\dfrac{\Gamma, x : \tau' \vdash M_1 : \tau\ \mathbf{(2)}}{\Gamma \vdash (\lambda x{:}\tau'.\, M_1) : \tau' \to \tau}\,\textsc{Abs} \qquad \Gamma \vdash V : \tau'\ \mathbf{(3)}}{\Gamma \vdash (\lambda x{:}\tau'.\, M_1)\ V : \tau\ (1)}\,\textsc{App}$$

The value-substitution Lemma applied to (2) and (3) gives the expected result.

Case $(\Lambda\alpha.V)\ \tau_0 \longrightarrow [\alpha \mapsto \tau_0]V$: By inversion of (1), we have $\Gamma, \alpha \vdash V : \tau_1$ (**4**) where τ is $[\alpha \mapsto \tau_0]\tau_1$. The type-substitution Lemma applied to (4) gives the expected result $\Gamma \vdash [\alpha \mapsto \tau_0]V : \tau$.

Case $E[M_0] \longrightarrow E[M_0']$: The hypothesis is $M_0 \longrightarrow M_0'$. Assume $\Gamma \vdash E[M_0] : \tau$. By compositionality, there is some type τ_0 and type variables $\vec{\alpha}$ such that $\Gamma, \vec{\alpha} \vdash M_0 : \tau_0$ (**5**) and for all M_0' such that $\Gamma, \vec{\alpha} \vdash M_0' : \tau_0$, we have $\Gamma \vdash E[M_0'] : \tau$. Therefore it suffices to show $\Gamma, \vec{\alpha} \vdash M_0' : \tau_0$, which holds by induction hypothesis applied to (5).

The classification lemma, which is a key to progress, is slightly modified to account for polymorphic types and constructed types. We need to state the lemma under an arbitrary set of type variables $\vec{\alpha}$ instead of the empty context—because evaluation is allowed under type abstractions.

Lemma 19 (Classification) *Assume* $\vec{\alpha} \vdash V : \tau$

- *If τ is an arrow type, then V is either a function* or a partial application of a constant to values*.*
- *If τ is a polymorphic type, then V is either a type abstraction of a value* or a partial application of a constant to types*.*
- *If τ is a constructed type, then V is constructed value.*

The last case can be refined by partitioning constructors into their associated type-constructor: If the top-most type constructor of τ is G, then V is a value constructed with a G-constructor.

The proof is similar to the one for simply-typed λ-calculus.

Progress is restated as follows:

Theorem 10 (Progress, strengthened) *A well-typed, irreducible closed term is a value: if $\vec{\alpha} \vdash M : \tau$ and $M \not\longrightarrow$, then M is some value V.*

The theorem has been strengthened, using a sequence of type variables $\vec{\alpha}$ for the typing context instead of the empty environment. It can then be proved by induction and case analysis on M, relying mainly on the classification lemma and the progress assumption for δ-reduction.

Proof: By induction on (the derivation of) M . Assume $\vec{\alpha} \vdash M : \tau$ and M is irreducible.

Case M is x: This is not possible since x is not well-typed in $\overline{\alpha}$.

Case M is c: Then M is a value (a fully applied constructor or a partially applied destructor), as expected.

Case M is $\lambda x{:}\tau.\,M_1$: Then M is a value, as expected.

Case M is $M_1\ M_2$: Then, $\vec{\alpha} \vdash M_1 : \tau_2 \to \tau_1$. and $\vec{\alpha} \vdash M_2 : \tau_2$. Since the left application is an evaluation context, M_1 is irreducible. Hence, by induction hypothesis, M_1 is a value. Since the right application of a value is an evaluation context, M_2 is irreducible. Hence, by induction hypothesis, M_2 is also a value. Since the application $M_1\ M_2$ itself cannot be reduced, M_1 is not a function. Since it has an arrow type, it follows from the classification lemma that it a partial application of a constant to values. Hence, M is itself the application of a constant to values. Since it cannot be reduced, it follows from the progress assumption for δ-rules that it is not a full application of a destructor. Hence, it is either a full application of a constructor or a partial application of a constant to values. In both cases, M is a value.

Case M is $\Lambda\beta.M_1$: Then, $\vec{\alpha}, \beta \vdash M_1 : \tau_1$. Since type abstraction is an evaluation context M_1 is irreducible. Hence, by induction hypothesis, M_1 is a value and so is M.

Case M is $M_1\ \tau_1$: Then, $\vec{\alpha} \vdash M_1 : \forall\alpha.\,\tau_2$ with τ equal to $[\alpha \mapsto \tau_1]\tau_2$. Since type application is an evaluation context, M_1 is irreducible. Hence, by induction hypothesis, M_1 is a value. Since M is irreducible M_1 is not a type abstraction. Since M_1 has a polymorphic type, it follows from the classification lemma that M_1 is an application of a constant c to types (as it is not a type abstraction). Since it is irreducible, it follows from the progress assumption for δ-rules that c is a destructor or the application is partial. In both cases M is a value.

Theorem 11 (Normalization) *Reduction terminates in pure System F.*

This is also true for arbitrary reductions and not just for call-by-value reduction. This is a difficult proof, which generalizes the proof method for the simply-typed λ-calculus. It is due to Girard (1972) (see also Girard et al. (1990)).

4.4 Type erasing semantics

We have presented the explicitly-typed variant of System F. In this section, we verify that this semantics is type erasing. Hence, there is an implicitly-typed presentation of System F.

4.4.1 Implicitly-typed System F

The implicitly-typed version of System F, can be defined as follows. The syntax of terms and their dynamic semantics are those of the untyped λ-calculus extended with constants. However, we only accept a subset of terms of the λ-calculus, retaining only those that are the type erasure of a term in F.

We write $\lceil \mathsf{F} \rceil$ for the set of implicitly-typed terms and F for the set of explicitly-typed terms. We use letters a, v, and e to range over implicitly-typed terms, values, and evaluation contexts, reusing the same notations as for the untyped λ-calculus.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \textsc{if-Var} \qquad \frac{}{\Gamma \vdash c : \Delta(c)}\ \textsc{if-Cst} \qquad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}\ \textsc{if-Abs} \qquad \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}\ \textsc{if-App}$$

$$\frac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall \alpha.\tau}\ \textsc{if-Tabs} \qquad \frac{\Gamma \vdash a : \forall \alpha.\tau}{\Gamma \vdash a : [\alpha \mapsto \tau_0]\tau}\ \textsc{if-Tapp}$$

Figure 4.2: Typing rules for explicitly-typed System F.

The set of terms may also be characterized by typing rules that operate directly on unannotated terms. These are obtained from the typing rules of F by dropping all type information in terms. They are presented in Figure 4.2. We use the prefix IF- to distinguish them from the typing rules for explicit System F.

Unsurprisingly, as a result of erasing type information in terms, the rules that introduce and eliminate the universal quantifier are no longer syntax-directed.

Remark 4 Notice that the explicit introduction of variable α in the premise of Rule TABS contains an implicit side condition $\alpha \# \Gamma$ due to the assumption on the formation of typing environments.

In implicitly-typed System F, as in ML, the introduction of type variables in typing context is often left implicit. (In some extensions of System F, type variables may carry a kind or a bound and must be explicitly introduced.) If we chose to do so, we would need an explicit side-condition on Rule TABS as follows:

$$\frac{\Gamma \vdash a : \tau \qquad \alpha \# \Gamma}{\Gamma \vdash a : \forall \alpha.\tau}\ \textsc{if-Tabs-Bis}$$

Omitting the side condition would lead to *unsoundness*. Below on the left-hand side is a type derivation for a *type cast* (*Obj.magic* in OCaml), which is equivalent to using an ill-formed context (on the right-hand side):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha_1 \vdash x : \alpha_1}\ \textsc{if-Var}}{x : \alpha_1 \vdash x : \forall \alpha_1.\alpha_1}\ \textsc{Broken if-Tabs}}{x : \alpha_1 \vdash x : \alpha_2}\ \textsc{if-Tapp}}{\varnothing \vdash \lambda x.\, x : \alpha_1 \to \alpha_2}\ \textsc{if-Abs}}{\varnothing \vdash \lambda x.\, x : \forall \alpha_1.\forall \alpha_2.\alpha_1 \to \alpha_2}\ \textsc{if-Tabs-Bis}
\qquad
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha_1, \alpha_1 \vdash x : \alpha_1}\ \textsc{Broken Var}}{x : \alpha_1 \vdash x : \forall \alpha_1.\alpha_1}\ \textsc{Broken Tabs}}{x : \alpha_1 \vdash x : \alpha_2}\ \textsc{Tapp}}{\varnothing \vdash \lambda x{:}\alpha_1.\, x : \alpha_1 \to \alpha_2}\ \textsc{Abs}}{\varnothing \vdash \Lambda \alpha_1.\Lambda \alpha_2.\lambda x{:}\alpha_1.\, x : \forall \alpha_1.\forall \alpha_2.\alpha_1 \to \alpha_2}\ \textsc{Tabs}$$

A good intuition is that a judgment $\Gamma \vdash a : \tau$ corresponds to the logical assertion $\forall \bar{\alpha}.(\Gamma \Rightarrow (a : \tau))$, where $\bar{\alpha}$ are the free type variables of the judgment, taken in any order. In this view, Tabs-Bis corresponds to the axiom:

$$\forall \alpha.(P \Rightarrow Q) \quad \equiv \quad P \Rightarrow (\forall \alpha.Q) \qquad \text{if } \alpha \# P$$

which without the side condition is obviously wrong.

The next lemma, states that the two definitions of $\lceil \mathsf{F} \rceil$—or, equivalently, the two type systems for implicitly-typed System F and explicitly type System F—coincide. The proof is immediate.

Lemma 20 *$\Gamma \vdash a : \tau$ in implicitly-typed System F if and only if there exists an explicitly-typed expression M whose erasure is a such that $\Gamma \vdash M : \tau$.*

For example, consider the term a_0 in $\lceil \mathsf{F} \rceil$ equal to $\lambda f x y.\,(f\ x,\ f\ y)$. A version that carries explicit type abstractions and annotations is:

$$\Lambda\alpha_1.\Lambda\alpha_2.\lambda f : \alpha_1 \to \alpha_2.\,\lambda x : \alpha_1.\,\lambda y : \alpha_1.\,(f\ x,\ f\ y)$$

Unsurprisingly, this term admits the polymorphic type:

$$\tau_1 \triangleq \forall\alpha_1.\forall\alpha_2.(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

Perhaps more surprising is the fact that this untyped term can be decorated in a different way:

$$\Lambda\alpha_1.\Lambda\alpha_2.\lambda f : \forall\alpha.\,\alpha \to \alpha.\,\lambda x : \alpha_1.\,\lambda y : \alpha_2.\,(f\ \alpha_1\ x,\ f\ \alpha_2\ y)$$

This term admits the polymorphic type:

$$\tau_2 \triangleq \forall\alpha_1.\forall\alpha_2.(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

This begs the question: which of the two types τ_1 or τ_2 is more general? Type τ_1 requires the second and third arguments to admit a common type, while type τ_2 requires the first argument to be polymorphic.

Exercise 30 (Distrib pair, disjoint types) *Find two terms a_1 and a_2 such that a_1 has type τ_1 but not type τ_2, and conversely for a_2. (Just give the terms a_1 and a_2, you do not have to prove well-typedness or ill-typedness.)* (Solution p. 86) □

This suggests that the two types are not comparable, that is, *neither one can be an instance of the other.*

Intuitively, one may think semantically of (*i.e.* interpret) a closed type as the set of terms of that type, and of instance as inclusion between types. With such a view in mind then τ_1 and τ_2 are indeed incomparable. This does not imply that a_0 does not have a principal type: there could exist a type τ_0 that contains a_0 and that is included in the intersection of (the interpretations of) τ_1 and τ_2. Indeed, one can do so in a richer system, such as System F^ω.

Exercise 31 (Distrib pair in F^ω) *Only if you know System F^ω: find a type τ_0 for a_0 in System F^ω that is more general than both τ_1 and τ_2,* i.e. *from which τ_1 and τ_2 can be obtained by rule* Inst-Gen. (Solution p. 86) □

4.4.2 Type instance

To reason formally, we must first define what it means for τ_2 to be an *instance* of τ_1—or, equivalently, for τ_1 to be *more general* than τ_2. Several definitions are possible. In System F, *to be an instance* is usually defined by the rule:

$$\textsc{Inst-Gen}\quad \frac{\vec{\beta} \# \forall \vec{\alpha}.\,\tau}{\forall \vec{\alpha}.\tau \le \forall \vec{\beta}.\,[\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

Notice that $\vec{\alpha}$ and $\vec{\beta}$ stands of (possibly empty) sequences of type variables. One can show that, if $\tau_1 \le \tau_2$, then any term that has type τ_1 has also type τ_2; that is, the following rule is *admissible*[2] in the implicitly-typed version:

$$\textsc{Sub}\quad \frac{\Gamma \vdash a : \tau_1 \qquad \tau_1 \le \tau_2}{\Gamma \vdash a : \tau_2}$$

Perhaps surprisingly, the rule is not *derivable*[3] in our presentation of System F. Although, we have the following derivation,

$$\textsc{Gen}^*\ \frac{\textsc{Inst}^*\ \dfrac{\Gamma, \vec{\beta} \vdash a : \forall \vec{\alpha}.\tau}{\Gamma, \vec{\beta} \vdash a : [\vec{\alpha} \mapsto \vec{\tau}]\tau}}{\Gamma \vdash a : \forall \vec{\beta}.[\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

the premise $\Gamma, \vec{\beta} \vdash a : \forall \alpha.\,\tau$ can only be justified from the assumption $\Gamma \vdash a : \forall \alpha.\,\tau$ by an application of weakening (the side condition $\vec{\beta} \# \forall \vec{\alpha}.\,\tau$ of rule Gen ensures that $\Gamma, \vec{\beta}$ is well-formed.) Otherwise, in context Γ alone, $\vec{\tau}$ would not necessarily be well-formed, as required by rule Gen.

However, in a version of System F that does not introduce type variables explicitly in Γ, then weakening of *type* variables would be built-in and implicit and the rule Sub would become derivable. (This shows that the notion of derivability is somewhat fragile as it depends on the presentation of the rules.)

We may also wonder what is the counter-part of the instance relation in explicitly-typed System F. Assume $\Gamma \vdash M : \tau_1$ and $\tau_1 \le \tau_2$. How can we see M with type τ_2? Since explicitly-typed terms have unique types, the term M of type τ_1 cannot itself also have type τ_2. However, we can wrap M with a *retyping context* that transforms a term of type τ_1 to one of type τ_2. Since $\tau_1 \le \tau_2$, the types τ_1 and τ_2 must be of the form $\forall \vec{\alpha}.\,\tau$ and $\forall \vec{\beta}.\,[\vec{\alpha} \mapsto \vec{\tau}]\tau$ where $\vec{\beta} \# \forall \vec{\alpha}.\,\tau$. *W.l.o.g*, we may assume that $\vec{\beta} \# \Gamma$ (6), as it may always be satisfied up to a renaming of bound variables $\vec{\beta}$. Then, we have the pseudo-derivation on the left-hand side (where the weakening lemma is used as a pseudo-typing rule Weakening), which can be

[2] A rule is *admissible* if adding the rule does not change the validity of judgments. That is, it may just allow for more derivations of already valid judgments.

[3] A rule is *derivable* if it can be replaced by a sub-derivation tree with the same premises and conclusion.

abbreviated by the admissible typing rule SUB given on the right-hand side.

$$\left.\dfrac{\dfrac{\dfrac{\Gamma \vdash M : \forall \vec{\alpha}.\,\tau \qquad (6)\,\dfrac{\vec{\beta} \mathrel{\#} \forall \vec{\alpha}.\,\tau}{\vec{\beta} \mathrel{\#} \Gamma}}{\Gamma, \vec{\beta} \vdash M : \forall \vec{\alpha}.\,\tau}\,\textsc{Weakening}}{\Gamma, \vec{\beta} \vdash M\,\vec{\tau} : [\vec{\alpha} \mapsto \vec{\tau}]\tau}\,\textsc{Tapp}^*}{\Gamma \vdash \Lambda\vec{\beta}.M\,\vec{\tau} : \forall \vec{\beta}.\,[\vec{\alpha} \mapsto \vec{\tau}]\tau}\,\textsc{Tabs}^*\right\}$$

Admissible rule:

$$\textsc{Sub}\;\dfrac{\Gamma \vdash M : \forall \vec{\alpha}.\,\tau \qquad \vec{\beta} \mathrel{\#} \forall \vec{\alpha}.\,\tau}{\Gamma \vdash \Lambda\vec{\beta}.M\,\vec{\tau} : \forall \vec{\beta}.\,[\vec{\alpha} \mapsto \vec{\tau}]\tau}$$

In F, we rather write subtyping as a judgment $\Gamma \vdash \tau_1 \leq \tau_2$ instead of the binary relation $\tau_1 \leq \tau_2$ to also mean $\Gamma \vdash \tau_1$ and $\Gamma \vdash \tau_2$ and so simultaneously keep track of the well-formedness of types.

In the previous example, the subtyping judgment $\Gamma \vdash \tau_1 \leq \tau_2$ has been witnessed by the wrapping context $\Lambda\vec{\beta}.[]\,\vec{\tau}$. Since this context is only composed of type abstractions and type applications, it changes the type of the term put in the hole without changing its behavior and it is called a *retyping context*. More generally, we may allow arbitrary wrappings of type abstractions and type applications around expressions. As in the example, they never change the type erasure. Retyping contexts are thus defined by the following grammar:

$$\mathcal{R} ::= [\,] \mid \Lambda\alpha.\mathcal{R} \mid \mathcal{R}\;\tau$$

(Notice that retyping contexts are arbitrarily deep here, by contrast with single-node evaluation contexts E defined earlier.)

We could also define a typing judgment $\Gamma \vdash \mathcal{R}[\tau_1] : \tau_2$ for retyping contexts as equivalent to $\Gamma, x : \tau_1 \vdash \mathcal{R}[x] : \tau_2$ whenever x does not appear in $\mathcal{R}$—or using primitive typing rules. Then, the following property holds by compositionality of typing: *if* $\Gamma \vdash M : \tau_1$ *and* $\Gamma \vdash \mathcal{R}[\tau_1] : \tau_2$, *then* $\Gamma \vdash \mathcal{R}[M] : \tau_2$.

We can now give another equivalent definition of subtyping, based on retyping contexts: $\Gamma \vdash \tau_1 \leq \tau_2$ *if and only if there exists a retyping context* $\mathcal{R}$ *such that* $\Gamma \vdash \mathcal{R}[\tau_1] : \tau_2$.

Notice that retyping contexts (*e.g.* type-instance) can only change topmost polymorphism. In particular, they cannot weaken the result types of functions or strengthen the types of their arguments.

4.4.3 Type containment in System F_η

Type containment is another, more expressive, syntactic notion of instance, introduced by Mitchell (1988), that can also transform inner parts of types. It can be defined syntactically by the following set of rules:

$$\textsc{Inst-Gen}\quad \frac{\vec{\beta} \# \forall \vec{\alpha}.\tau}{\forall \vec{\alpha}.\tau \le \forall \vec{\beta}.[\vec{\alpha} \mapsto \vec{\tau}]\tau} \qquad \textsc{Distributivity}\quad \forall \alpha.(\tau_1 \to \tau_2) \le (\forall \alpha.\tau_1) \to (\forall \alpha.\tau_2) \qquad \textsc{Congruence-}\to\quad \frac{\tau_2 \le \tau_1 \qquad \tau_1' \le \tau_2'}{\tau_1 \to \tau_1' \le \tau_2 \to \tau_2'}$$

$$\textsc{Congruence-}\forall\quad \frac{\tau_1 \le \tau_2}{\forall \alpha.\tau_1 \le \forall \alpha.\tau_2} \qquad \textsc{Transitivity}\quad \frac{\tau_1 \le \tau_2 \qquad \tau_2 \le \tau_3}{\tau_1 \le \tau_3}$$

With this larger instance relation, Rule Sub is no longer admissible—as it allows to type more terms. However, it remains sound. That is, adding Rule Sub as a primitive typing rule does not break type soundness. The resulting type system is known as System F_η, since it is also the closure of System F by η-expansion; that is, a term is in System F_η if and only if it is the η-conversion of a term in System F.

Exercise 32 *1) Show that $\forall \alpha.\tau \equiv \tau$ whenever $\alpha \notin \mathsf{ftv}(\tau)$. 2) Show that rule* Distributivity *can be replaced by the weaker rule:*

$$\textsc{Distrib-Right}\quad \frac{\alpha \notin \mathsf{ftv}(\tau_1)}{\forall \alpha.(\tau_1 \to \tau_2) \le \tau_1 \to (\forall \alpha.\tau_2)}$$

(Solution p. 87) □

One may wonder what System F_η brings to System F that it does not already have. Consider the identity function *id* in $\lceil \mathsf{F} \rceil$; it has type $\forall \alpha.\alpha \to \alpha$ but also many other incomparable types. For example, it has type $(\forall \alpha.\alpha) \to \forall \alpha.\alpha \to \alpha$— even though a function of that type can never be applied, as there is no value of type $\forall \alpha.\alpha$ that could be passed as argument; it also has the more interesting type $\forall \alpha.(\forall \alpha.\alpha \to \alpha) \to (\alpha \to \alpha)$. While these types are incomparable in $\lceil \mathsf{F} \rceil$, they become comparable in System F_η. For example, in System F_η, we have:

$$\tau_{id} \quad \le \quad \left(\begin{array}{lll} (\forall \alpha.\alpha) \to (\forall \alpha.\alpha) & \le & (\forall \alpha.\alpha) \to \tau_{id} \\ \forall \beta.(\beta \to \beta) \to (\beta \to \beta) & \le & \forall \beta.\tau_{id} \to (\beta \to \beta) \end{array} \right) \quad \le \quad \forall \beta.(\forall \alpha.\alpha) \to (\beta \to \beta)$$

The type $\forall \alpha.\alpha \to \alpha$ is actually a principal type for *id* in System F_η. Similarly, the function *ch* defined below has a principal type in System F_η:

$$ch \quad \triangleq \quad \lambda x.\lambda y.\mathsf{if}\ M\ \mathsf{then}\ x\ \mathsf{else}\ y \quad : \quad \forall \beta.\beta \to \beta \to \beta$$

Still, many expressions do not have most general types in System F_η. To see the difficulty, consider the application *chid* of *ch* to *id*. How can it be typed? If we keep *id* polymorphic, then *chid* has type $(\forall \alpha.\alpha \to \alpha) \to (\forall \alpha.\alpha \to \alpha)$, say τ_1; if, on the opposite, we instantiate *id*, then *chid* has type $\forall \alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$, say τ_2—as in ML where type schemes are automatically instantiated when used. These two types are incomparable in System F. Although, we have $\tau_1 \le \tau_2$ in System F_η (as witnessed by the coercion context

$\lambda x : \forall \alpha. \alpha \to \alpha. \Lambda \alpha. ([\tau_2]\ \alpha)\ (x\ \alpha))$ and can thus give *chid* the type τ_2 and still used it at type τ_1, this is more by chance than the general case: If we replace *ch* by ch_3, which chooses between three arguments, then $ch_3\ id$ does not have a principal type in System F_η.

System F_η increases the expressiveness of System F by enriching its type instance relation—without modifying the language of types (and other typing rules than Sub).

To obtain even more principal types, Le Botlan and Rémy (2009) have suggested that the language of types should be enriched with a new form of quantification $\forall \alpha \geq \tau_1. \tau_2$ to mean, intuitively, the set of types $[\alpha \mapsto \tau]\tau_2$ when τ ranges over the set of instances of τ_1. This internalizes the instance relation within the language of types. This allows to give *chid* the type $\forall (\beta \geq \forall \alpha. \alpha \to \alpha). \beta \to \beta$ and recovering $(\forall \alpha. \alpha \to \alpha) \to (\forall \alpha. \alpha \to \alpha)$ and $\forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha)$ by choosing particular instances of $\forall \alpha. \alpha \to \alpha$ for β. By contrast with System F_η, this approach also works for the more general example of $ch_3\ id$.

The language MLF has been design for partial type reconstruction where programs are partially annotated. The user need only to provide the types of parameters of functions that are used polymorphically. The type systems is setup to implicitly use available polymorphism but never guess polymorphism. Available polymorphism comes either from type generalization as in ML or from user-provided type annotations. Every expression has a principal type—according to the given type annotations. See (Le Botlan and Rémy, 2009; Rémy and Yakobowski, 2008) for details.

4.4.4 A definition of principal typings

A typing of an expression M is a pair Γ, τ such that $\Gamma \vdash M : \tau$. Ideally, a type system should satisfy the *principal typings* property (Wells, 2002):

> *Every well-typed term M admits a principal typing – one whose* instances *are exactly the typings of M.*

Whether this property holds depends on a definition of *instance*. The more liberal the instance relation, the more hope there is of having principal typings.

The instance relations we have previously considered are defined syntactically. The absence of principal typings with respect to a syntactic definition of instance may result from a bad choice of the instance relation. To avoid arbitrariness, Wells (2002) introduced a more *semantic* notion of instance. He notes that, once a type system is fixed, a most liberal notion of instance can be defined, a posteriori, by:

> *A typing θ_1 is more general than a typing θ_2 if and only if every term that admits θ_1 admits θ_2 as well.*

This is the largest reasonable notion of instance: $\leq$ is defined as the largest relation such that a subtyping principle is admissible.

This definition can be used to prove that a system does *not* have principal typings, under *any* reasonable definition of "instance". Then, which systems have principal typings? The *simply-typed λ-calculus has principal typings,* with respect to a substitution-based notion of instance (See lesson on type inference). Wells (2002) shows that *neither System F nor System F_η have principal typings.* It was shown earlier that *System F_η's instance relation is undecidable* (Wells, 1995; Tiuryn and Urzyczyn, 2002) and that *type inference for both System F and System F_η is undecidable* (Wells, 1999).

There are still a few positive results. Some systems of *intersection types* have principal typings (Wells, 2002) – but they are very complex and have yet to see a practical application.

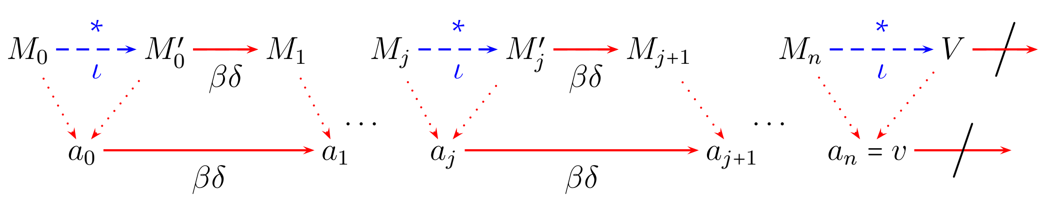
A weaker property is to have *principal types.* Given an environment Γ and an expression M is there a type τ for M in Γ such that all other types of M in Γ are instances of τ. Damas and Milner's type system (coming up next) does not have *principal typings* but it has *principal types* and *decidable type inference.*

4.4.5 Type soundness for implicitly-typed System F

Subject reduction and progress imply the soundness of the *explicitly*-typed version of *System* F. What about the *implicitly*-typed version? Can we reuse the soundness proof for the explicitly-typed version? Can we pullback subject reduction and progress from F to $\lceil \mathsf{F} \rceil$?

For progress, given a well-typed term a in $\lceil \mathsf{F} \rceil$, can we find a term M in F whose erasure is a and such that M is a value or reduces, and so conclude that a is a value or reduces? For subject reduction, given a term a_1 of type τ in $\lceil \mathsf{F} \rceil$ that reduces to a_2, can we find a term M_1 in F whose erasure is a_1 and show that M_1 reduces to a term M_2 whose erasure is a_2 to conclude that the type of a_2 is the type of a_1? In both cases, this reasoning requires a *type-erasing* semantics. We claimed that the explicitly-typed System F has an erasing semantics. We now verify it.

There is a difference with the simply-typed λ-calculus because the reduction of type applications on explicitly-typed terms is dropped by type erasure, hence the two reductions cannot coincide *exactly*. The way to formalize this is to split reduction steps into $\beta\delta$-steps corresponding to β or δ rules that must be preserved by type erasure, and ι-steps corresponding to the reduction of type applications that disappear during type erasure. This can be summarized in the following diagram:



We say that we establish a *bisimulation* between reduction on typed-terms and their erasure *up to ι-steps*. The bisimulation can be decomposed into a direct and a inverse simulation.

Lemma 21 (Direct simulation) *The reduction in* F *is simulated in* $\lceil \mathsf{F} \rceil$ *up to* ι*-steps. Assume* $\Gamma \vdash M : \tau$*. Then:*
1) If $M \longrightarrow_\iota M'$*, then* $\lceil M \rceil = \lceil M' \rceil$
2) If $M \longrightarrow_{\beta\delta} M'$*, then* $\lceil M \rceil \longrightarrow_{\beta\delta} \lceil M' \rceil$

The inverse direction is more delicate to state, since type erasure is not bijective: there are usually many expressions of F whose type erasure is a given expression in $\lceil \mathsf{F} \rceil$.

Lemma 22 (Inverse simulation) *Assume* $\Gamma \vdash M : \tau$ *and* $\lceil M \rceil \longrightarrow a$*. Then, there exists a term* M' *such that* $M \longrightarrow_\iota^* \longrightarrow_{\beta\delta} M'$ *and* $\lceil M' \rceil = a$*.*

Of course, the semantics can only be type erasing if δ-rules do not themselves depend on type information. First, we need δ-reduction to be defined on type erasures. We may prove the theorem directly for some concrete examples of δ-reduction.

However, keeping δ-reduction abstract is preferable to avoid repeating the same reasoning many times. Then, we must assume that it is such that type erasure establishes a bisimulation for δ-reduction taken alone.

Assumption on δ. We assume that for any explicitly-typed term M of the form $d\,\tau_1 \ldots \tau_j\, V_1 \ldots V_k$ such that $\Gamma \vdash M : \tau$, both of the following properties hold:

(Direct bisimulation) If $M \longrightarrow_\delta M'$, then $\lceil M \rceil \longrightarrow_\delta \lceil M' \rceil$.

(Inverse bisimulation) If $\lceil M \rceil \longrightarrow_\delta a$, then there exists M' such that $M \longrightarrow_\delta M'$ and a is the type-erasure of M'.

In most cases, the assumption on δ-reduction is obvious to check. Notice however, that in general the δ-reduction on untyped terms is larger than the projection of δ-reduction on typed terms, because it pattern matches on the shapes of values but ignoring types. However, if we restrict δ-reduction to implicitly-typed terms, then it usually coincides with the projection of reduction of explicitly-typed terms.

Exercise 33 *Consider the explicitly-typed System* F *with pairs of the exercise 24 (p. 57). Add pairs in the untyped* λ*-calculus. Show that* δ*-reduction in the untyped* λ*-calculus is larger than the image of the* δ*-reduction in the explicitly-typed calculus. Verify that type erasure is a bisimulation for* δ*-reduction.* (Solution p. 87) □

The direct simulation (Lemma 21) is straightforward to establish. (Details of the proof p. 88)

The inverse simulation is slightly more delicate because there may be many antecedents of a given type erasure. We use a few easy helper lemmas to keep the proof clearer.

Lemma 23

(1) A term that erases to $\bar{e}[a]$ *is of the form* $\bar{E}[M]$ *where* $\lceil \bar{E} \rceil$ *is* $\bar{e}$ *and* $\lceil M \rceil$ *is* a*; moreover, we may assume that* M *does not start with a type abstraction nor a type application.*

(2) If $\bar{E}$ erases to the empty context then $\bar{E}$ is a retyping context $\mathcal{R}$.

(3) If $\mathcal{R}[M]$ is in ι-normal form, then $\mathcal{R}$ is of the form $\Lambda\vec{\alpha}.[]\,\vec{\tau}$.

The main helper lemma is :

Lemma 24 (Inversion of type erasure) *Assume $\lceil M \rceil = a$*

- *If a is x, then M is of the form $\mathcal{R}[x]$*
- *If a is c, then M is of the form $\mathcal{R}[c]$*
- *If a is $\lambda x.\, a_1$, then M is of the form $\mathcal{R}[\lambda x{:}\tau.\, M_1]$ with $\lceil M_1 \rceil = a_1$*
- *If a is $a_1\, a_2$, then M is of the form $\mathcal{R}[M_1\; M_2]$ with $\lceil M_i \rceil = a_i$*

The proof is by an induction on M.

Lemma 25 (Inversion of type erasure for well-typed values) *Assume $\Gamma \vdash M : \tau$ and M is ι-normal. If $\lceil M \rceil$ is a value v, then M is a value V. Moreover,*

- *If v is $\lambda x.\, a_1$, then V is $\Lambda\vec{\alpha}.\lambda x{:}\tau.\, M_1$ with $\lceil M_1 \rceil = a_1$.*
- *If v is a partial application $c\, v_1\ \ldots v_n$ then V is $\mathcal{R}[c\,\vec{\tau}\; V_1\ \ldots V_n]$ with $\lceil V_i \rceil = v_i$.*

The proof is by induction on M. It uses the inversion of type erasure, then analysis of the typing derivation to restrict the form of retyping contexts. (Details of the proof p. 88)

Corollary 26 *Let M be a well-typed term in ι-normal form whose erasure is a*

- *If a is $(\lambda x.\, a_1)\; v$ then M is be of the form $\mathcal{R}[(\lambda x{:}\tau.\, M_1)\; V]$, with $\lceil M_1 \rceil$ equal to a_1 and $\lceil V \rceil$ equal to v.*

- *If a is a full application $c\, v_1\ \ldots v_n$ then M is of the form $\mathcal{R}[c\,\vec{\tau}\; V_1\ \ldots V_n]$ with $\lceil V_i \rceil = v_i$.*

(Proof p. 88)

We may now prove inverse simulation. It suffices to prove it when M is ι-normal. The general case follows, since one may first ι-reduce M to a normal form M_0, while preserving typings, thanks to subject reduction and type erasure; the lemma can then be applied to M_0 instead of M. Notice that this reasoning relies on the termination of ι-reduction. Indeed, if ι-reduction could diverge, it is unlikely that the semantics would be type erasing.

Termination of ι-reduction follows indirectly from the termination of reduction in System F. Its direct proof is also immediate, as ι-reduction strictly decreases the number of type abstractions.

Proof (inverse simulation): The proof is by induction on the reduction of $\lceil M \rceil$. We assume M is in ι-normal form.

Case $\lceil M \rceil$ *is* $(\lambda x.\, a_1)\ v$: By Corollary 26, M is of the form $\mathcal{R}[(\lambda x\!:\!\tau_1.\, M_1)\ V]$. Since $\mathcal{R}$ is an evaluation context, M reduces to $\mathcal{R}[[x \mapsto V]M_1]$ whose erasure is $[x \mapsto v]a_1$, *i.e.* a.

Case $\lceil M \rceil$ *is* $e[a_1]$ *and* $a_1 \longrightarrow a_2$: By Lemma 23, M is of the form $\bar{E}[M_1]$ where $\lceil \bar{E} \rceil$ is $\bar{e}$ and $\lceil M_1 \rceil$ is a_1. By compositionality (Lemma 18), M_1 is well-typed. Since M is ι-normal and $\bar{E}$ is an evaluation, M_1 is also ι-normal. By induction hypothesis, M_1 reduces in one $\beta\delta$ step to a term M_2 whose erasure is a_2. Hence, by Context, M reduces in one $\beta\delta$-step to the term $\bar{E}[M_2]$ whose erasure is $\bar{e}[a_2]$, *i.e.* a.

Case $\lceil M \rceil$ *is a full application* $(d\ v_1 \ldots\ v_n)$ *and reduces to* a: By Corollary 26, M is of the form $\mathcal{R}[M_0]$ where M_0 is $d\,\vec{\tau}\ V_1\ \ldots V_n$ a $\lceil V_i \rceil$ is v_i. Since $\lceil M_0 \rceil \longrightarrow a$, by the inverse assumption for δ-rules, there exists M_0' such that $M_0 \longrightarrow_\delta M_0'$ and $\lceil M_0' \rceil$ is a. Let M' be $\mathcal{R}[M_0']$. Since $\mathcal{R}$ is an evaluation context, we have $M \longrightarrow_\delta M'$ and $\lceil M' \rceil$ is a.

We may now easily transpose subject reduction and progress from the implicitly-typed version to the implicitly-typed version of System F.

Theorem 12 (Type soundness for implicitly-typed System F)
Progress and subject reduction holds in implicitly-typed System F.

Proof: Assume that $\Gamma \vdash a_1 : \tau$. By Lemma 20, there exists a term M_1 such that $\Gamma \vdash M_1 : \tau$. and $\lceil M_1 \rceil$ is a_1.

Progress: Let M_2 be the ι-normal form of M_1. By direct simulation, $\lceil M_2 \rceil$ is a. By subject reduction, we have $\Gamma \vdash M_2 : \tau$. By progress in F, either M_2 $\beta\delta$-reduces and so does a, by direct simulation (Lemma 21) or M_2 is a value and so is its erasure a_1 (by observation).

Subject reduction: Assume $a_1 \longrightarrow a_2$. By inverse simulation (Lemma 22), there exists a term M_2 such that $M_1 \longrightarrow_\iota^* \longrightarrow_{\beta\delta} M_2$ and $\lceil M_2 \rceil$ is a_2. By subject reduction in F, we have $\Gamma \vdash M_2 : \tau$. By Lemma 20, we have $\Gamma \vdash a_2 : \tau$, as expected.

Remarks The design of advanced typed systems for programming languages is usually done in explicitly-typed version, with a type-erasing semantics in mind, but this is not always checked in details (and sometimes not even made very clear). While the direct simulation is usually straightforward, the inverse simulation is often harder. As the type system gets more complicated, reduction at the level of types also gets more involved. It is important and not always obvious that type reduction terminates *and* is rich enough to never block reductions that could occur in the type erasure.

For example, Crétin and Rémy (2012) extend System F_η with abstraction over retyping functions, but keep the type systems bridled to preserve the type erasure semantics.

Bisimulation is a standard technique to show that compilation preserves the semantics given in small-step style. For example, it is *heavily* used in the CompCert project (Leroy, 2006) to prove the correctness of a compiler from C to assembly code, using the Coq proof assistant. The compilation from C to assembly code is decomposed into a chain of transformation using a dozen of successive intermediate languages; each of the transformation is then proved to be semantic preserving using bisimulation techniques.

4.5 Polymorphism and references

In this chapter, we have just shown how to extend simply-typed λ-calculus with polymorphism. In the previous chapter we have shown how to extend simply-typed λ-calculus with references. Can these extensions be combined together?

When adding references, we noted that type soundness relies on the fact that *every reference cell (or memory location) has a fixed type.* Otherwise, if a location had two types $\mathsf{ref}\ \tau_1$ and $\mathsf{ref}\ \tau_2$, one could store a value of type τ_1 and read back a value of type τ_2. Hence, it should also be unsound if a location could have type $\forall\alpha.\,\mathsf{ref}\ \tau$ (where α appears in τ) as it could then be specialized to both types $\mathsf{ref}\ [\alpha \mapsto \tau_1]\tau$ and $\mathsf{ref}\ [\alpha \mapsto \tau_2]\tau$. By contrast, a location ℓ can have type $\mathsf{ref}\ (\forall\alpha.\,\tau)$: this says that ℓ stores values of polymorphic type $\forall\alpha.\,\tau$, but ℓ, as a value, is viewed with the monomorphic type $\mathsf{ref}\ (\forall\alpha.\,\tau)$.

4.5.1 A counter example

Still, if System F is naively extended with references, it allows the construction of polymorphic references, which breaks subject reduction:

$\mathsf{let}\ y : \forall\alpha.\,\mathsf{ref}\ (\alpha \to \alpha) =$	(2) Abstracts α and binds ℓ to y of type $\forall\alpha.\,\mathsf{ref}\ (\alpha \to \alpha)$
$\quad \Lambda\alpha.\mathsf{ref}\ (\alpha \to \alpha)\ (\lambda z{:}\alpha.\,z)$	(1) Creates and returns a location ℓ of type $\mathsf{ref}\ (\alpha \to \alpha)$ bound to the identity function $\lambda z{:}\alpha.\,z$ of type $\alpha \to \alpha$
in	
$(y\ \mathsf{bool}) := (\mathsf{bool} \to \mathsf{bool})\ \mathit{not};$	(3) Writes the location at type $\mathsf{bool} \to \mathsf{bool}$
$!(\mathsf{int} \to \mathsf{int})\ (y\ (\mathsf{int}))\ 1 \quad / \quad \varnothing$	(4) Reads it back at type $\mathsf{int} \to \mathsf{int}$

$$\xrightarrow{*} \quad \mathit{not}\ 1 \quad / \quad \ell \mapsto \mathit{not}$$

The program is well-typed, but reduces to the stuck expression "*not* 1". So what went wrong? As described on the right-hand side, the fault is that the location is written at type bool and read back at type int. This is permitted because the location has a polymorphic type $\forall\alpha.\,\mathsf{ref}\ \alpha \to \alpha$. So this must be wrong. Indeed, the first reduction step uses the following rule (where V is $\lambda x{:}\alpha.\, x$ and τ is $\alpha \to \alpha$).

$$\textsc{Context}\ \frac{\mathsf{ref}\ \tau\ V \,/\, \varnothing \longrightarrow \ell \,/\, \ell \mapsto V}{\Lambda\alpha.\mathsf{ref}\ \tau\ V \,/\, \varnothing \longrightarrow \Lambda\alpha.\ell \,/\, \ell \mapsto V}$$

While we have

$$\alpha \vdash \mathsf{ref}\ \tau\ V \,/\, \varnothing : \mathsf{ref}\ \tau \qquad \text{and} \qquad \alpha \vdash \ell \,/\, \ell \mapsto V : \mathsf{ref}\ \tau$$

We have

$$\vdash \Lambda\alpha.\mathsf{ref}\ \tau\ V \,/\, \varnothing : \forall\alpha.\, \mathsf{ref}\ \tau \qquad \text{but not} \qquad \vdash \Lambda\alpha.\ell \,/\, \ell \mapsto V : \forall\alpha.\, \mathsf{ref}\ \tau$$

Hence, the context case of subject reduction breaks.

The typing derivation of $\Lambda\alpha.\ell$ requires a store typing Σ of the form $\ell : \tau$ and a derivation of the form (according to Rule Loc given below, page 4.5.2):

$$\textsc{Tabs}\ \frac{\Sigma, \alpha \vdash \ell : \mathsf{ref}\ \tau}{\Sigma \vdash \Lambda\alpha.\ell : \forall\alpha.\, \mathsf{ref}\ \tau}$$

However, the typing context Σ, α is ill-formed as α appears free in Σ. Instead, a well-formed premise should bind α earlier as in $\alpha, \Sigma \vdash \ell : \mathsf{ref}\ \tau$, but then, Rule Tabs cannot be applied.

By contrast, the expression $\mathsf{ref}\ \tau\ V$ is pure, so Σ may be empty:

$$\textsc{Tabs}\ \frac{\alpha \vdash \mathsf{ref}\ \tau\ V : \mathsf{ref}\ \tau}{\varnothing \vdash \Lambda\alpha.\mathsf{ref}\ \tau\ V : \forall\alpha.\, \mathsf{ref}\ \tau}$$

The expression $\Lambda\alpha.\ell$ is correctly rejected as ill-typed, so $\Lambda\alpha.\mathsf{ref}\ \tau\ V$ should also be rejected. There is a fix to the bug known as this mysterious slogan:

> *One must not abstract over a type variable that* might, after evaluation of the term, *enter the store typing.*

Indeed, this is what happens in our example. The type variable α which appears in the type of V is abstracted in front of $\mathsf{ref}\ \tau\ V$. When $\mathsf{ref}\ \tau\ V$ reduces, $\alpha \to \alpha$ becomes the type of the fresh location ℓ, which appears in the new store typing. This is all well and good, but *how* do we enforce this slogan?

In the context of ML, a number of rather complex historic approaches have been followed: see Leroy (1992) for a survey. Then came Wright (1995), who suggested an amazingly simple solution, known as the *value restriction:* only *value forms* can be abstracted over.

$$\textsc{TAbs}\ \frac{\Gamma, \alpha \vdash U : \tau}{\Gamma \vdash \Lambda\alpha.U : \forall\alpha.\tau} \qquad\qquad \textsc{Value forms:}\ U ::= x \mid V \mid \Lambda\alpha.U \mid U\ \tau$$

The problematic proof case *vanishes*, as we now never $\beta\delta$-reduce under type abstraction, only ι-reduction is possible. Subject reduction holds again. Let us prove it.

4.5.2 Internalizing configurations

A configuration $M \,/\, \mu$ is an expression M in a memory μ. Intuitively, the memory can be viewed as a recursive extensible mutable record. The configuration $M \,/\, \mu$ may be viewed as the recursive definition (of values) $\mathsf{let\ rec}\ \ m : \Sigma = \mu\ \mathsf{in}\ [\overline{\ell} \mapsto m.\overline{\ell}]M$ where Σ is a store typing for μ. The store typing rules are coherent with this view. For instance, allocation of a reference is a reduction of the form:

$$\begin{array}{ll} & \mathsf{let\ rec}\ \ m : \Sigma \qquad\quad = \mu \qquad\quad \mathsf{in}\ E[\mathsf{ref}\ \tau\ V] \\ \longrightarrow & \mathsf{let\ rec}\ \ m : \Sigma, \ell : \tau = \mu, \ell \mapsto v\ \mathsf{in}\ E[m.\ell] \end{array}$$

For this transformation to preserve well-typedness, it is clear that the evaluation context E must not bind any type variable appearing in τ; otherwise, we are violating the scoping rules.

Let use clarify the typing rules for configurations:

$$\frac{\vec{\alpha} \vdash M : \tau \qquad \vec{\alpha} \vdash \mu : \Sigma}{\vec{\alpha} \vdash M \,/\, \mu : \tau}\ \textsc{Config} \qquad\qquad \frac{\forall \ell \in \mathsf{dom}(\mu), \quad \vec{\alpha}, \Sigma, \varnothing \vdash \mu(\ell) : \Sigma(\ell)}{\vec{\alpha} \vdash \mu : \Sigma}\ \textsc{Store}$$

Closed configurations must be typed in an environment composed of type variables. No new type variables is never introduced during reduction. These type variables may appear in the store typing during reduction, there are thus placed in front the store typing and cannot be generalized.

Judgments are now of the form $\vec{\alpha}, \Sigma, \Gamma \vdash M : \tau$ although we may see $\vec{\alpha}, \Sigma, \Gamma$ as a whole typing context Γ'. For locations, we need a new context formation rule:

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \tau \qquad \ell \notin \mathsf{dom}(\Gamma)}{\vdash \Gamma, \ell : \tau}\ \textsc{WfEnvLoc}$$

This allows locations to appear anywhere. However, in a derivation of a closed term, the typing context will always be of the form $\vec{\alpha}, \Sigma, \Gamma$ where Σ only binds locations (to arbitrary types) and Γ does not bind locations.

The typing rule for memory locations (where Γ is of the form $\vec{\alpha}, \Sigma, \Gamma'$) is:

$$\textsc{Loc} \quad \Gamma \vdash \ell : \mathsf{ref}\ \Gamma(\ell)$$

In System F, typing rules for references need not be primitive. We may instead treat them as constants of the following types:

$$\mathsf{ref} \colon\ \forall \alpha.\, \alpha \to \mathsf{ref}\ \alpha \qquad\qquad (!)\ :\ \forall \alpha.\, \mathsf{ref}\ \alpha \to \alpha \qquad\qquad (:=)\ :\ \forall \alpha.\, \mathsf{ref}\ \alpha \to \alpha \to \mathsf{unit}$$

They are all destructors (event ref) with the obvious arities.

The δ-rules are adapted to carry explicit type parameters:

$$\mathsf{ref}\ \tau\ V \,/\, \mu \longrightarrow \ell \,/\, \mu[\ell \mapsto V] \qquad \text{if } \ell \notin \mathsf{dom}(\mu)$$

$$\ell \coloneqq (\tau)\ V \,/\, \mu \longrightarrow () \,/\, \mu[\ell \mapsto V] \qquad\qquad !\tau\ \ell \,/\, \mu \longrightarrow \mu(\ell) \,/\, \mu$$

Type soundness can now be stated as

Lemma 27 *δ-rules preserve well-typedness of closed configurations.*

Theorem 13 (Subject reduction) *Reduction of closed configurations preserves well-typedness.*

Lemma 28 *A well-typed closed configuration M/μ where M is a full application of constants* ref*,* (!)*, and* (:=) *to types and values can always be reduced.*

Theorem 14 (Progress) *A well typed irreducible closed configuration M/μ is a value.*

As a sanity check, the problematic program is now syntactically ill-formed:

$$\begin{array}{l} \mathsf{let}\ y : \forall\alpha.\,\mathsf{ref}\ (\alpha \to \alpha) = \Lambda\alpha.\mathsf{ref}\ (\alpha \to \alpha)\ (\lambda z{:}\alpha.\,z)\ \mathsf{in} \\ \quad (y\ \mathsf{bool}) \coloneqq (\mathsf{bool} \to \mathsf{bool})\ not; \\ \quad !(\mathsf{int} \to \mathsf{int})\ (y\ (\mathsf{int}))1 \end{array}$$

Indeed, $\mathsf{ref}\ (\alpha \to \alpha)\ (\lambda z{:}\alpha.\,z)$ is not a value, but the application of a unary destructor to a value, so the expression $\Lambda\alpha.\mathsf{ref}\ (\alpha \to \alpha)\ (\lambda z{:}\alpha.\,z)$ is not allowed.

Consequences With the value restriction, some pure programs become ill-typed, even though they were well-typed in the absence of references. This style of introducing references in System F (or in ML) is *not a conservative extension.*

Assuming functions *map* and *id* of respective types $\forall\alpha.\,\mathsf{list}\ \alpha \to \mathsf{list}\ \alpha$ and $\forall\alpha.\,\alpha \to \alpha$, the expression $\Lambda\alpha.map\ \alpha\ (id\ \alpha)$ is now ill-typed. A common work-around is to perform a manual η-expansion $\Lambda\alpha.\lambda y{:}\mathsf{list}\ \alpha.\, map\ \alpha\ (id\ \alpha)\ y$. However, in the presence of side effects, η-expansion is *not* semantics preserving, so this must not be done blindly.

In practice, the value restriction can be slightly relaxed by enlarging the class of value forms to a syntactic category of so-called *non-expansive terms*—terms whose evaluation will definitely not allocate new reference cells. Non-expansive terms form a strict superset of value forms. Garrigue (2004) relaxes the value restriction in a more subtle way, which is justified by a subtyping argument. For instance, the following expressions may be well-typed:

- $\Lambda\alpha.((\lambda x{:}\tau.\,U)\ U)$ because the inner expression is non-expansive;
- $\Lambda\alpha.(\mathsf{let}\ x : \tau = U\ \mathsf{in}\ U)$, which is its syntactic sugar;
- $\mathsf{let}\ x : \forall\alpha.\,\mathsf{list}\ \alpha = \Lambda\alpha.(M_1\ M_2)\ \mathsf{in}\ M$ because α appears only positively in the type of $eappM_1M_2$.

OCaml implements both refinements.

In fact, $\Lambda\alpha.M$ need only be forbidden when α appears negatively in the type of some exposed expansive terms where exposed subterms are those that do not appear under some λ-abstraction. For instance, the expression

$$\mathsf{let}\ x : \forall\alpha.\, \mathsf{int} \times (\mathsf{list}\ \alpha) \times (\alpha \to \alpha) = \Lambda\alpha.(\mathsf{ref}\ (1+2),\ (\lambda x{:}\alpha.\,x)\ \mathit{Nil},\ \lambda x{:}\alpha.\,x)\ \mathsf{in}\ M$$

may be well-typed because α appears only in the type of the non-expansive exposed expressions $\lambda x{:}\alpha.\,x$ and positively in the type of expansive expression $(\lambda x{:}\alpha.\,x)\ \mathit{Nil}$.

(This refinement is not implemented in OCaml, though.)

Remark Experience has shown that *the value restriction is tolerable.* Even though it is not conservative, the search for better solutions has been pretty much abandoned.

In a type-and-effect system (Lucassen and Gifford, 1988; Talpin and Jouvelot, 1994), or in a type-and-capability system (Charguéraud and Pottier, 2008), the type system indicates which expressions may allocate new references, and at which type. There, the value restriction is no longer necessary—but these systems are heavy. However, if one extends a type-and-capability system with a mechanism for *hiding* state, which remains useful even in those systems, the need for the value restriction re-appears.

Pottier and Protzenko (2012) are designing a language Mezzo where mutable states is tracked quite precisely, with permissions, ownership, linear types that even enable a reference to even change the type of its values over time, which is called *strong update.*

4.6 Damas and Milner's type system

Damas and Milner's type system Milner (1978) offers a restricted form of polymorphism, while avoiding the difficulties associated with type inference in System F. This type system is at the heart of Standard ML, OCaml, and Haskell.

The idea behind the definition of ML is to make a small extension of simply-typed λ-calculus that enables to factor out several occurrences of the same subexpression a_1 in a term of the form $[x \mapsto a_1]a_2$ using a let-binding form $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ so as to avoid code duplication.

Expressions of the simply-typed λ-calculus are extended with a primitive let-binding, which can also be viewed as a way of annotating some redexes $(\lambda x.\, a_2)\ a_1$ in the source program. This actually provides a simple intuition behind Damas and Milner's type system: *a closed term has type τ if and only if its* let-normal form *has type τ in simply-typed λ-calculus.* A term's let-normal form is obtained by iterating the following rewrite rule, in any context:

$$\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \quad\longrightarrow\quad a_1; [x \mapsto a_1]a_2$$

Notice that we use a sequence starting with a_1 and not just $[x \mapsto a_1]a_2$. This is to enforce well-typedness of a_1 in the pathological case where x does not appear free in a_2. If we disallow this pathological case (*e.g.* well-formedness could require that x always occurs in a_2) then we could just use the more intuitive rewrite rule:

$$\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \quad \longrightarrow \quad [x \mapsto a_1]a_2$$

This intuition suggests type-checking and type inference algorithms. However, these algorithms are *not practical,* because they have *intrinsic* exponential complexity; and separate compilation prevents reduction to let-normal forms.

In the following, we study a direct presentation of Damas and Milner's type system, which does not involve let-normal forms. It is *practical*, because it leads to an efficient type inference algorithm (presented in chapter §8); and it supports separate compilation.

4.6.1 Definition

The language ML is usually presented in its implicitly-typed version, where *terms* are given by:

$$a ::= x \mid c \mid \lambda x.\, a \mid a\ a \mid \mathsf{let}\ x = a\ \mathsf{in}\ a \mid \ldots$$

The *let* construct is no longer sugar for a β-redex but a primitive form that will be typed especially.

The language of types lies between those for simply-typed λ-calculus and System F; it is stratified between *types* and *type schemes.* The syntax of *types* is that of simply-typed λ-calculus, but a separate category of *type schemes* is introduced:

$$\tau ::= \alpha \mid \tau \to \tau \mid \ldots \qquad\qquad \sigma ::= \tau \mid \forall \alpha.\, \sigma$$

All quantifiers must appear in *prenex position,* so type schemes are less expressive than System-F types. We often write $\forall \vec{\alpha}.\, \tau$ as a short hand for $\forall \alpha_1.\, \ldots \forall \alpha_n.\, \tau$. When viewed as a subset of System F, one must think of *type schemes* are the primary notion of types, of which *types* are a subset.

An ML typing context Γ binds program variables to *type schemes.* In the implicitly-typed presentation, type variables are often introduced implicitly and not part of Γ. However, we keep below the equivalent presentation where type variables are declared in Γ. Judgments now take the form $\Gamma \vdash a : \sigma$. Types form a subset of type schemes, so type environments and judgments can contain types too.

The standard, non-syntax-directed presentation of ML is given in Figure 4.3. Rule Let moves a type scheme into the environment, which Var can exploit. Rule Abs and App are unchanged. *λ-bound variables receive a monotype.* Rule Gen and Inst are as in implicitly-typed System F, except that *type variables are instantiated with monotypes.*

For example, here is a type derivation that exploits polymorphism (writing Γ for f :

$$\text{IML-VAR}\ \ \Gamma \vdash x : \Gamma(x) \qquad \text{IML-CST}\ \ \Gamma \vdash c : \Delta(c) \qquad \text{IML-ABS}\ \ \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau} \qquad \text{IML-APP}\ \ \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\text{IML-LET}\ \ \frac{\Gamma \vdash a_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash a_2 : \sigma_2}{\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \sigma_2} \qquad \text{IML-GEN}\ \ \frac{\Gamma, \alpha \vdash a : \sigma}{\Gamma \vdash a : \forall \alpha.\, \sigma} \qquad \text{IML-INST}\ \ \frac{\Gamma \vdash a : \forall \alpha.\, \sigma}{\Gamma \vdash a : [\alpha \mapsto \tau]\sigma}$$

Figure 4.3: Typing rules for ML

$\forall \alpha.\, \alpha \to \alpha$.) for an implicitly-typed term (omitting the IML- prefix of typing rules):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha, z : \alpha \vdash z : \alpha}\,\text{VAR}}{\alpha \vdash \lambda z.\, z : \alpha \to \alpha}\,\text{ABS}}{\varnothing \vdash \lambda z.\, z : \forall \alpha.\, \alpha \to \alpha}\,\text{GEN} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \forall \alpha.\, \alpha \to \alpha}\,\text{VAR}}{\Gamma \vdash f : \mathsf{int} \to \mathsf{int}}\,\text{INST}}{\Gamma \vdash f\ 0 : \mathsf{int}}\,\text{APP} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \forall \alpha.\, \alpha \to \alpha}\,\text{VAR}}{\Gamma \vdash f : \mathsf{bool} \to \mathsf{bool}}\,\text{INST}}{\Gamma \vdash f\ \mathsf{true} : \mathsf{bool}}\,\text{APP}}{\Gamma \vdash (f\ 0,\ f\ \mathsf{true}) : \mathsf{int} \times \mathsf{bool}}\,\text{PAIR}}{\varnothing \vdash \mathsf{let}\ f = \lambda z.\, z\ \mathsf{in}\ (f\ 0,\ f\ \mathsf{true}) : \mathsf{int} \times \mathsf{bool}}\,\text{LET}$$

Notice that Rule GEN is used above LET (on the left-hand side), and INST is used below VAR. In fact, we will see below that every type derivation can be transformed into one of this form.

As a counter-example, the term $\lambda f.\, (f\ 0,\ f\ \mathsf{true})$ is *ill-typed*. Indeed, as it contains no "let" construct, it is type-checked exactly as in simply-typed λ-calculus, where it is ill-typed, because f must be assigned a type τ that must simultaneously be of the form $\mathsf{int} \to \tau_1$ and $\mathsf{bool} \to \tau_2$, but there is no such type. Recall that this term is well-typed in implicitly-typed System F because f can be assigned, for instance, the polymorphic type $\forall \alpha.\, \alpha \to \alpha$.

While we rather use implicitly-typed terms in programs, we usually prefer to use an explicitly-typed presentation of ML in proofs. We thus identify a subset of terms of System F whose type erasure coincide with terms of ML. The subset of terms is defined by the following syntax:

$$M \in e\mathsf{ML} ::= x \mid c \mid \lambda x{:}\tau.\, M \mid M\ M \mid \Lambda \alpha. M \mid M\ \tau \mid \mathsf{let}\ x : \sigma = M\ \mathsf{in}\ M \ldots$$

where τ and σ are ML-types and type schemes and not arbitrary System-F types. The typing rules for explicitly-typed terms are given on Figure 4.4.

These are restrictions of the typing rules of System-F to terms and types of ML. Therefore, if $\Gamma \vdash_{e\mathsf{ML}} M : \sigma$ then $\Gamma \vdash_{\mathsf{F}} M : \sigma$. In particular, explicitly-typed terms of ML have unique typing derivations—and actually unique types—as in System-F.

Unfortunately, the converse is not true—when M is syntactically in ML and Γ and σ are well-formed in eML, of course. Hence, the relation $\vdash_{e\mathsf{ML}}$ cannot be defined as the restriction of $\vdash_{\mathsf{F}}$ to ML environments terms and type schemes.

Exercise 34 *Find a term M that is syntactically in eML and a type scheme σ such that*

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \textsc{eml-Var} \qquad \frac{}{\Gamma \vdash c : \Delta(c)}\ \textsc{eml-Cst} \qquad \frac{\Gamma, x : \tau_0 \vdash M : \tau}{\Gamma \vdash \lambda x{:}\tau_0.\, M : \tau_0 \to \tau}\ \textsc{eml-Abs} \qquad \frac{\Gamma \vdash M_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash M_1\ M_2 : \tau_1}\ \textsc{eml-App}$$

$$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \sigma_2}{\Gamma \vdash \mathsf{let}\ x : \sigma = M_1\ \mathsf{in}\ M_2 : \sigma_2}\ \textsc{eml-Let} \qquad \frac{\Gamma, \alpha \vdash M : \sigma}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\,\sigma}\ \textsc{eml-Tabs} \qquad \frac{\Gamma \vdash M : \forall\alpha.\,\sigma}{\Gamma \vdash M\ \tau : [\alpha \mapsto \tau]\sigma}\ \textsc{eml-Tapp}$$

Figure 4.4: Typing rules for eML (explicitly-typed ML)

$$\frac{\Gamma, \vec{\alpha} \vdash Q : \tau}{\Gamma \vdash \Lambda\vec{\alpha}.Q : \forall\vec{\alpha}.\,\tau}\ \textsc{xml-Tabs} \qquad \frac{\Gamma, x : \tau_0 \vdash Q : \tau}{\Gamma \vdash \lambda x{:}\tau_0.\, Q : \tau_0 \to \tau}\ \textsc{xml-Abs} \qquad \frac{\Gamma \vdash Q_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash Q_2 : \tau_2}{\Gamma \vdash Q_1\ Q_2 : \tau_1}\ \textsc{xml-App}$$

$$\frac{\Gamma, \vec{\alpha} \vdash Q_1 : \tau_1 \qquad \Gamma, x : \forall\vec{\alpha}.\,\tau_1 \vdash Q_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x : \forall\vec{\alpha}.\,\tau_1 = \Lambda\vec{\alpha}.Q_1\ \mathsf{in}\ Q_2 : \tau_2}\ \textsc{xml-LetGen} \qquad \frac{\forall\vec{\alpha}.\,\tau = \Gamma(x)}{\Gamma \vdash x\ \vec{\tau} : [\vec{\alpha} \mapsto \vec{\tau}]\tau}\ \textsc{xml-VarInst} \qquad \frac{\forall\vec{\alpha}.\,\tau = \Delta(c)}{\Gamma \vdash c\ \vec{\tau} : [\vec{\alpha} \mapsto \vec{\tau}]\tau}\ \textsc{xml-CstInst}$$

Figure 4.5: Typing rules for xML

$\Gamma \vdash_{\mathsf{F}} M : \sigma$ *holds but* $\Gamma \vdash_{e\mathsf{ML}} M : \sigma$ *does not hold.* *(Solution p. 89)* □

4.6.2 Syntax-directed presentation

Explicitly-typed terms of ML have unique typing derivations—and actually unique types—as in System-F. By contrast with explicitly-typed terms, implicitly-typed terms have several types, since parameters of functions are not annotated, but also several typing derivations, since places for type abstraction and type applications are not specified either, much as in System F.

Interestingly, there is a syntax-directed presentation of implicitly-typed ML terms where the shape of typing derivations is entirely determined by the term and is thus unique. Taking the explicitly-typed view, this amounts to restricting the source terms so that there is no choice for placing type abstraction and type applications.

Let xML be the subset of explicitly-typed ML defined by the following grammar

$$\begin{array}{rcl} N \in x\mathsf{ML} & ::= & \Lambda\vec{\alpha}.Q \\ Q & ::= & x\ \vec{\tau} \mid Q\ Q \mid \lambda x{:}\tau.\, Q \mid \mathsf{let}\ x : \sigma = N\ \mathsf{in}\ Q \end{array}$$

where τ here ranges over simple types and such that all type variables are fully instantiated. That is, we request that the arity of $\vec{\tau}$ in $x\ \vec{\tau}$ be the arity of $\vec{\alpha}$ in the type scheme $\forall\vec{\alpha}.\,\tau$ assigned to the variable x. In particular, all Q-terms are typed with simple types.

Specializing the typing rules of eML (Figure 4.4) to the syntax of xML gives the typing rules of xML on Figure 4.5. By construction, terms of xML are a syntactic subset of terms

$$\text{norm-Var}\quad \frac{\forall\vec{\alpha}.\tau = \Gamma(x)}{\Gamma \vdash x : \forall\vec{\alpha}.\tau \Rightarrow \Lambda\vec{\alpha}.x\ \vec{\alpha}}$$

$$\text{norm-Tabs}\quad \frac{\Gamma, \alpha \vdash M : \sigma \Rightarrow N}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma \Rightarrow \Lambda\alpha.N}$$

$$\text{norm-Tapp}\quad \frac{\Gamma \vdash M : \forall\alpha.\sigma \Rightarrow \Lambda\alpha.N}{\Gamma \vdash M\ \tau : [\alpha \mapsto \tau]\sigma \Rightarrow [\alpha \mapsto \tau]N}$$

$$\text{norm-Cst}\quad \frac{\forall\vec{\alpha}.\tau = \Delta(c)}{\Gamma \vdash c : \forall\vec{\alpha}.\tau \Rightarrow \Lambda\vec{\alpha}.c\ \vec{\alpha}}$$

$$\text{norm-Let}\quad \frac{\Gamma \vdash M_1 : \sigma_1 \Rightarrow N_1 \qquad \Gamma, x : \sigma_1 \vdash M_2 : \forall\vec{\alpha}.\tau \Rightarrow \Lambda\vec{\alpha}.Q \qquad \vec{\alpha} \# N_1, \sigma_1}{\Gamma \vdash \mathsf{let}\ x : \sigma_1 = M_1\ \mathsf{in}\ M_2 : \forall\vec{\alpha}.\tau \Rightarrow \Lambda\vec{\alpha}.\mathsf{let}\ x : \sigma_1 = N_1\ \mathsf{in}\ Q}$$

$$\text{norm-App}\quad \frac{\Gamma \vdash M_1 : \tau_2 \to \tau_1 \Rightarrow Q_1 \qquad \Gamma \vdash M_2 : \tau_2 \Rightarrow Q_2}{\Gamma \vdash M_1\ M_2 : \tau_1 \Rightarrow Q_1\ Q_2}$$

$$\text{norm-Abs}\quad \frac{\Gamma, x : \tau_0 \vdash M : \tau \Rightarrow Q}{\Gamma \vdash \lambda x{:}\tau_0.\,M : \tau_0 \to \tau \Rightarrow \lambda x{:}\tau_0.\,Q}$$

Figure 4.6: Normalization of ML derivations

of $e\mathsf{ML}$. By construction, we also have if $\Gamma \vdash_{x\mathsf{ML}} M : \sigma$ then $\Gamma \vdash_{e\mathsf{ML}} M : \sigma$.

Conversely, we wish to show that any term M typable in $e\mathsf{ML}$ can be mapped to a term N typable in $x\mathsf{ML}$ that has the same type erasure. For this purpose, we define on Figure 4.6 a normalization judgment $\Gamma \vdash M : \sigma \Rightarrow N$ by inference rules, which can also be read as an algorithm that performs:

- Type η-expansion of every occurrence of a variable according to the arity of its type scheme (Rule Var). This ensures that every occurrence of a type variable will be fully specialized—hence assigned a monomorphic type.
- Strong ι-reduction, *i.e.* type β-reduction (Rule Tapp): this cancels type applications of type abstractions. As a result, elaborated terms do not contain any ι-redex.

The translation is well-defined for all $e\mathsf{ML}$ terms, since it follows the structure of the typing derivation in $e\mathsf{ML}$. Formally, if $\Gamma \vdash_{e\mathsf{ML}} M : \sigma$ holds then $\Gamma \vdash M : \sigma \Rightarrow N$ holds. The proof is by induction on M and all cases are obvious.

Moreover, if $\Gamma \vdash M : \sigma$ holds, then $\Gamma \vdash_{x\mathsf{ML}} N : \sigma$ also holds and M and N have the same erasure. The proof is also by induction on M. The preservation of erasure is immediate. The only non obvious cases for well-typedness of N are Norm-Tapp, which performs strong ι-reduction and uses type substitution (Lemma 17), and Norm-Let, which extrudes type abstractions.

Another way to look at the normalization of terms is as a rewriting of the typing derivations so that all applications of Inst come immediately after Var and all applications of Gen come immediately above rule Let or at the bottom of the derivation—as imposed by the grammar of $x\mathsf{ML}$ terms where Q-terms can only have monomorphic types.

$$\frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}\ \textsc{ml-Abs} \qquad \frac{\Gamma \vdash a_2 : \tau_2 \qquad \Gamma \vdash a_1 : \tau_2 \to \tau_1}{\Gamma \vdash a_1\ a_2 : \tau_1}\ \textsc{ml-App}$$

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \vec\alpha \ \#\ \Gamma \qquad \Gamma, x : \forall \vec\alpha.\, \tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau_2}\ \textsc{ml-LetGen} \qquad \frac{\forall \vec\alpha.\, \tau = \Gamma(x)}{\Gamma \vdash x : [\vec\alpha \mapsto \vec\tau]\tau}\ \textsc{ml-VarInst} \qquad \frac{\forall \vec\alpha.\, \tau = \Delta(c)}{\Gamma \vdash c : [\vec\alpha \mapsto \vec\tau]\tau}\ \textsc{ml-VarCst}$$

Figure 4.7: Syntax-directed rules for ML

In summary, any term of eML can be rearranged as a term of xML with the same type erasure. By dropping type information in terms of xML, we then obtain a syntax-directed presentation of implicitly-typed ML, called sML:

$$\frac{\Gamma, \vec\alpha \vdash M : \tau}{\Gamma \vdash \Lambda \vec\alpha . M : \forall \vec\alpha.\, \tau}\ \textsc{xml-Tabs} \qquad \frac{\Gamma, x : \tau_0 \vdash a : \tau}{\Gamma \vdash \lambda x.\, a : \tau_0 \to \tau}\ \textsc{sml-Abs} \qquad \frac{\Gamma \vdash a_2 : \tau_2 \qquad \Gamma \vdash a_1 : \tau_2 \to \tau_1}{\Gamma \vdash a_1\ a_2 : \tau_1}\ \textsc{sml-App}$$

$$\frac{\Gamma, \vec\alpha \vdash a_1 : \tau_1 \qquad \Gamma, x : \forall \vec\alpha.\, \tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \tau_2}\ \textsc{sml-LetGen} \qquad \frac{\forall \vec\alpha.\, \tau = \Gamma(x)}{\Gamma \vdash x : [\vec\alpha \mapsto \vec\tau]\tau}\ \textsc{sml-VarInst} \qquad \frac{\forall \vec\alpha.\, \tau = \Delta(c)}{\Gamma \vdash c : [\vec\alpha \mapsto \vec\tau]\tau}\ \textsc{sml-CstInst}$$

Then, the judgments $\Gamma \vdash_{\mathsf{ML}} a : \tau$ and $\Gamma \vdash_{s\mathsf{ML}} a : \tau$ are equivalent.

However, for type inference, we rather use the equivalent presentation in Figure 4.7 called iML (or the inference type system) where type variables are not explicitly declared in the typing context—hence, the side condition for generalization on rule Let.

In this final system, type substitution (Lemma 17), which we will use for type inference, can be restated as follows:

Lemma 29 (Type Substitution) *Typings are stable by substitution.*
If $\Gamma \vdash a : \tau$ *then* $\varphi\Gamma \vdash a : \varphi\tau$. *for any substitution* φ.

4.6.3 Type soundness for ML

Since ML is a subset of $\lceil \mathsf{F} \rceil$, which has been proved sound, we know that ML is sound, *i.e.* that ML programs cannot go wrong. This also implies that progress holds in ML. However, we do not know whether subject reduction holds for ML. Indeed, ML expressions could reduce to System F expressions that are not in the ML subset. Most proofs of subject reduction for implicitly-typed ML work directly with implicitly-typed terms. See for instance (Wright and Felleisen, 1994; Pottier and Rémy, 2005).

Subject-reduction in eML The proof of subject reduction follows the same schema as for System F (Theorem 9). The main part of the proof works almost unchanged. However, it uses auxiliary lemmas (inversion, permutation, weakening, type substitution, term substitution, compositionality) that all need to be rechecked, since those lemmas conclude with typing judgments in F that may not necessarily hold in $e\mathsf{ML}$. Unsurprisingly, all proofs can be easily adjusted.

An indirect proof reusing subject-reduction in System F We also present an indirect proof that reuses subject reduction and progress in System F and the syntax-directed presentation of ML.

To establish subject-reduction in ML, let a_1 be an implicitly-typed ML term such that both $\vec{\alpha} \vdash_{\mathsf{ML}} a_1 : \sigma$ and $a_1 \longrightarrow a_2$ hold. There exists an explicitly-typed term M_1 such that $\vec{\alpha} \vdash_{e\mathsf{ML}} M_1 : \sigma$ and $\lceil M_1 \rceil = a_1$. By normalization, we may elaborate M_1 into a term N_1 of $x\mathsf{ML}$ such that $\vec{\alpha} \vdash_{x\mathsf{ML}} N_1 : \sigma$ and the $\lceil N_1 \rceil = \lceil M_1 \rceil$. Moreover, N_1 is by construction ι-normal. Since $x\mathsf{ML}$ is a subset of System F, we have $\vec{\alpha} \vdash_{\mathsf{F}} N_1 : \sigma$. By inverse simulation in System F (Lemma 22), there exists N_2 in F whose type erasure is a_2 and such that $N_1 \longrightarrow_\beta N_2$ (since N_1 is ι-normal). We show below that there exists a strong ι-reduction M_2 of N_2 that is in $x\mathsf{ML}$ and such that $\vec{\alpha} \vdash_{x\mathsf{ML}} N_2 : \sigma$. Therefore, we have $\vec{\alpha} \vdash_{e\mathsf{ML}} M_2 : \sigma$ and since the type erasure of M_2 is that of N_2, *i.e.* a_2, we have $\vec{\alpha} \vdash_{\mathsf{ML}} a_2 : \sigma$, as expected.

It thus remains to check that given a term N_1 such that $\Gamma \vdash_{x\mathsf{ML}} N_1 : \sigma$ and $N_1 \longrightarrow_\beta N_2$, there exists a term M_2 in $x\mathsf{ML}$ that is a strong ι-reduction of N_2 and such that $\Gamma \vdash_{x\mathsf{ML}} N_2 : \sigma$. This can be decomposed into the existence of M_2 and type preservation by strong ι-reduction.

The β-reduction step may occur in any evaluation context and is one of two forms. If it is a normal β-reduction:

$$(\lambda x{:}\tau.\, Q)\; V \longrightarrow [x \mapsto V]Q$$

it preserves syntactic membership in $e\mathsf{ML}$, because since x is bound to a type and its occurrences in M cannot be specialized. However, if it is a let-reduction

$$\mathsf{let}\ x : \forall \vec{\alpha}.\, \tau = V\ \mathsf{in}\ Q \longrightarrow [x \mapsto V]Q$$

then occurrences of x in Q, which are of the form $x\ \vec{\tau}$, become $V\ \vec{\tau}$ and may contain ι-redexes—which are not allowed in $x\mathsf{ML}$. Fortunately, V is necessarily of the form $\Lambda\vec{\alpha}.V'$ where the arity of $\vec{\alpha}$ is equal to that of $\vec{\tau}$. Hence, we may immediately perform a sequence of ι-reduction that brings the term back into $x\mathsf{ML}$ and in ι-normal form. Notice however that this ι-redex is not in general in a call-by-value evaluation context. Indeed, x may appear under an abstraction in M. Hence, this is a strong reduction step.

For type reduction, we need to ensure that strong ι-reduction is also type-preserving. This is an easy auxiliary proof—but not a consequence of subject reduction, which we have only proved for reduction in call-by-value evaluation contexts.

4.7 Ommitted proofs and answers to exercises

Solution of Exercise 24

As in the case where pairs are primitive, we introduce one constructor $(\cdot,\cdot)$ of arity 2 and and two destructors proj_1 and proj_2 of arity 1, with the following types in Δ

$$\begin{array}{rl} Pair: & \forall\alpha_1.\,\forall\alpha_2.\,\alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2 \\ proj_i: & \forall\alpha_1.\,\forall\alpha_2.\,\alpha_1 \times \alpha_2 \to \alpha_i \end{array}$$

and the two reduction rules:

$$proj_i\ \tau_1\ \tau_2\ (Pair\ \tau_1'\ \tau_2'\ V_1\ V_2) \longrightarrow V_i \qquad (\delta_i)$$

We then only need to verify that δ_i preserves types and ensure progress.

Case Type preservation: Assume that $\Gamma \vdash proj_i\ \tau_1\ \tau_2\ (Pair\ \tau_1'\ \tau_2'\ V_1\ V_2) : \tau$. By inversion, it must be the case that τ is equal to τ_i and $\Gamma \vdash V_i : \tau_i$ holds, which ensures our goal $\Gamma \vdash V_i : \tau$.

Case Progress: Assume that $\Gamma \vdash M : \tau$ and M is of the form $proj_i\ \tau_1\ \tau_2\ V$. By the inversion lemma, τ must be a product type $\tau_1 \times \tau_2$ such that $\Gamma \vdash V : \tau_1 \times \tau_2$. By the classification lemma, V must be a pair, *i.e.* of a form $Pair\ \tau_1\ \tau_2\ V_1\ V_2$. Hence, M reduces to V_i by δ_i. ■

Solution of Exercise 25

We introduce a new type constructor bool, two nullary constructors true and false of type bool and one ternary destructor ifcase of type $\forall\alpha..\mathsf{bool} \to \alpha \to \alpha \to \alpha$ with two reduction rules:

$$\mathsf{ifcase}\ \tau\ \mathsf{true}\ V_1\ V_2 \longrightarrow V_1 \qquad\qquad \mathsf{ifcase}\ \tau\ \mathsf{false}\ V_1\ V_2 \longrightarrow V_2$$

This extension is sound.

However, it defines a strict semantics for the conditional, while a lazy semantics is expected: indeed, since the destructor is ternary, $\mathsf{ifcase}\ \tau\ V_0\ [\,]\ M$ and $\mathsf{ifcase}\ \tau\ V_0\ V_1\ [\,]$ are evaluations contexts, which allows to reduce the two branches before selecting the right one.

An easy fix is to introduce $\mathsf{iflazy}\ \tau\ M_0\ M_1\ M_2$ as syntactic sugar for

$$(\mathsf{ifcase}\ \ \tau\ M_0\ (\lambda() : \mathsf{unit}.\, M_1)\ (\lambda() : \mathsf{unit}.\, M_2))\ ()$$

and exposing it to the user, while hiding the primitive ifcase from the user. ■

Solution of Exercise 26

1) We introduce a new unary type constructor list ; two constructors $\mathsf{Nil}\ \cdot$ and Cons of types $\forall\alpha.\,\mathsf{list}\ \alpha$ and $\forall\alpha.\,\alpha \to \mathsf{list}\ \alpha \to \mathsf{list}\ \alpha$; and one destructor $\mathsf{matchlist}\ \cdot\cdot\cdot$ of type:

$$\forall\alpha\beta.\,\mathsf{list}\ \alpha \to \beta \to (\alpha \to \mathsf{list}\ \alpha \to \beta) \to \beta$$

with the two reduction rules:

$$\begin{array}{l}\mathsf{matchlist}\ \tau\ (\mathsf{Nil}\ \tau')\ V_n\ V_c \longrightarrow V_n \\ \mathsf{matchlist}\ \tau\ (\mathsf{Cons}\ \tau'\ V_h\ V_t)\ V_n\ V_c \longrightarrow V_c\ V_h\ V_t\end{array}$$

2) Ommitted. ∎

Solution of Exercise 27

In ML, we may define the datatype:

```
type any = Fold of (any → any)
```

This can be simulated by adding a new type *any*, a constructor *Any* and a destructor *unany* of types $(\mathsf{any} \to \mathsf{any}) \to \mathsf{any}$ and $\mathsf{any} \to (\mathsf{any} \to \mathsf{any})$, respectively, with the following reduction rule:

$$\mathsf{unfold}\ (\mathsf{Fold}\ V) \longrightarrow V \qquad \delta_{any}$$

Let us check soundness of this extension:

Case Type preservation: Assume that $\Gamma \vdash \mathsf{unfold}\ (\mathsf{Fold}\ V) : \tau$. By inversion, we known that τ is $\mathsf{any} \to \mathsf{any}$ and that $\Gamma \vdash V : \mathsf{any} \to \mathsf{any}$, which shows our goal $\Gamma \vdash V : \tau$.

Case Progress: Assume that $\Gamma \vdash \mathsf{unfold}\ V : \tau$. By inversion, τ must be $\mathsf{any} \to \mathsf{any}$ and $\Gamma \vdash V : \mathsf{any}$ holds. By classification, V must be $\mathsf{Fold}\ V_0$. Hence, $\mathsf{unfold}\ V$ reduces.

The fixpoint can be defined in the λ-calculus (or in ML with recursive types) as :

```
let zfix g = (fun x → x x) (fun z → g (fun v → z z v))
```

We may implement zfix in ML without recursive tyes as:

```
type any = Fold of (any → any);;
let unfold (Fold x) = x;;
let zfix g =
  (fun x → unfold (x (Fold x)))
    (fun z → Fold (g (fun v → unfold ((unfold z) z) v)));;
```

∎

Proof of Lemma 16

Assume $\Gamma, x : \tau_0, \Gamma' \vdash M : \tau$ (**1**) and $\Gamma \vdash M_0 : \tau_0$ (**2**). We show $\Gamma, \Gamma' \vdash [x \mapsto M_0]M : \tau$ (**3**). by induction and cases on M and applying the inversion lemma to (1).

Case M is x: By (1), it must be the case that τ is equal to τ_0. Hence, the goal (3) is $\Gamma, \Gamma' \vdash M_0 : \tau_0$, which follows from the hypothesis (2) by weakening.

Case M is y when $y \neq x$: By (1), $y : \tau$ is in $\mathsf{dom}(\Gamma, x : \tau_0, \Gamma')$, actually in $\mathsf{dom}(\Gamma, \Gamma')$, since y is not x. Hence the goal (3) follows by Rule VAR.

Case M is c: By (1), $c : \tau$ is in Δ. Hence, the goal (3) follows by Rule VAR.

Case M is $\lambda y : \tau_1 .\, M_1$: By (1), τ is of the form $\tau_2 \to \tau_1$ and $\Gamma, x : \tau_0, \Gamma', y : \tau_2 \vdash M_1 : \tau_1$ holds. By induction hypothesis, we have $\Gamma, \Gamma', y : \tau_2 \vdash [x \mapsto M_0]M_1 : \tau_1$. By rule Abs, we have $\Gamma, \Gamma' \vdash \lambda y{:}\tau_2 .\, [x \mapsto M_0]M_1 : \tau_1$, which is the goal (3).

Case M is $\Lambda\alpha.M_1$: By (1), we have $\Gamma, x : \tau, \Gamma', \alpha \vdash M_1 : \tau_1$ and τ is equal to $\forall\alpha.\,\tau_1$. By induction hypothesis, we have $\Gamma, \Gamma', \alpha \vdash [x \mapsto M_0]M_1 : \tau_1$. By rule Tabs, we have $\Gamma \vdash \Lambda\alpha.[x \mapsto M_0]M_1 : \forall\alpha.\,\tau_1$, which is the goal (3).

Case M is $M_1\ M_2$ or M is $M_1\ \tau_1$: Immediate.

Proof of Lemma 17

The proof is by induction on M using inversion of the typing derivation of $\Gamma, \alpha, \Gamma' \vdash M : \tau$ (**1**). We write θ for $[\alpha \mapsto \tau]$. We must show $\Gamma, \theta\Gamma' \vdash \theta M : \theta\tau$ (**2**).

Case M is x: By (1), we have $x : \tau$ must be in Γ, α, Γ'. If $x : \tau$ is in Γ, then by well-formedness of types, α does not appear free in τ. Hence $\theta\tau$ is τ and $x : \theta\tau$ is in Γ. Otherwise, $x : \tau$ is in Γ' and $x : \theta\tau$ is in $\theta\Gamma'$. In both cases, $x : \theta\tau$ is in $\Gamma, \theta\Gamma'$. Hence, the conclusion follows by Rule Var.

Case M is c: By (1), we have $c : \tau$ is in Δ and τ is closed. Hence $\theta\tau$ is equal to τ and $c : \theta\tau$ is still in Δ. Thus, the conclusion (2) follows by Rule Const.

Case M is $\lambda x : \tau_0 .\, M_1$: By (1) and inversion, we have $\Gamma, \alpha, \Gamma', x : \tau_0 \vdash M_1 : \tau_1$ where τ is $\tau_0 \to \tau_1$. By induction hypothesis, $\Gamma, \theta(\Gamma', x : \tau_0) \vdash M_1 : \tau_1$, *i.e.* $\Gamma, \theta\Gamma', x : \theta\tau_0 \vdash \theta M_1 : \theta\tau_1$. By Rule Abs, we have $\Gamma, \theta\Gamma' \vdash \lambda x{:}\theta\tau_0 .\, \theta M_1 : \theta\tau_0 \to \theta\tau_1$, *i.e.* (2).

Case M is $\Lambda\beta.M_1$: By (1) and inversion, we have $\Gamma, \alpha, \Gamma', \beta \vdash M_1 : \tau_1$ where τ is $\forall\beta.\,\tau_1$. By induction hypothesis, we have $\Gamma, \theta(\Gamma', \beta) \vdash \theta M_1 : \theta\tau_1$, which is equal to $\Gamma, \theta\Gamma', \beta \vdash \theta M_1 : \theta\tau_1$. By rule Tabs, we $\Gamma, \theta\Gamma' \vdash \Lambda\beta.\theta M_1 : \forall\beta.\,\theta\tau$, *i.e.* *i.e.* (2).

Case M is $M_1\ M_2$ or M is $M_1\ \tau_1$: Immediate.

Solution of Exercise 30

Take, for instance, $\lambda f.\,\lambda x.\,\lambda y.\,(f\ y,\ f\ x)$ for a_1 (notice the inverse order of fields in the pair) and $\lambda f.\,\lambda x.\,\lambda y.\,(f\ (f\ x),\ f\ (f\ y))$ for a_2. ∎

Solution of Exercise 31

Choose, for instance,

$$\Lambda\alpha_1.\Lambda\alpha_2.\Lambda\varphi_1.\Lambda\varphi_2.(\forall\alpha.\,\varphi_1(\alpha) \to \varphi_2(\alpha)) \to \varphi_1(\alpha_1) \to \varphi_1(\alpha_2) \to \varphi_2(\alpha_1) \times \varphi_2(\alpha_1)$$

for τ_0. We recover τ_1 by choosing the constant functions $\lambda\alpha.\,\alpha_i$ for φ_i and τ_2 by choosing the identity $\lambda\alpha.\,\alpha$ for both φ_1 and φ_2. ∎

Solution of Exercise 32

1) Both directions follow from rule Inst-Gen, just applying the substitution $\alpha \mapsto \alpha$ for the direct implication and just generalizing over α for the reverse.

2) Rule Distrib-Right is a particular case of Distributivity, indeed. Assuming $\alpha \notin \mathsf{ftv}(\tau_1)$, and using the previous equivalence (**1**), we have

$$\dfrac{\overset{\text{Distributivity}}{\forall\alpha.\,(\tau_1 \to \tau_2) \le (\forall\alpha.\,\tau_1) \to (\forall\alpha.\,\tau_2)} \qquad \dfrac{\tau_1 \le \forall\alpha.\,\tau_1 \ (1)}{(\forall\alpha.\,\tau_1) \to (\forall\alpha.\,\tau_2) \le \tau_1 \to (\forall\alpha.\,\tau_2)}\ \text{Congruence-}\to}{\forall\alpha.\,(\tau_1 \to \tau_2) \le \tau_1 \to (\forall\alpha.\,\tau_2)}\ \text{Trans}$$

Conversely, we have the following derivation:

$$\text{Distrib-Right}\ \dfrac{\text{Congruence-}\forall\ \dfrac{\text{Congruence-}\to\ \dfrac{\text{Inst-Gen}\ \forall\alpha.\,\tau_1 \le \tau_1}{\tau_1 \to \tau_2 \le (\forall\alpha.\,\tau_1) \to \tau_2}}{\forall\alpha.\,(\tau_1 \to \tau_2) \le \forall\alpha.\,(\forall\alpha.\,\tau_1) \to \tau_2}}{\forall\alpha.\,(\tau_1 \to \tau_2) \le (\forall\alpha.\,\tau_1) \to (\forall\alpha.\,\tau_2)}$$

■

Solution of Exercise 33

We extend the λ-calculus with a binary constructor $Pair$ and two unary destructors $proj_i$ for i in $\{1, 2\}$ with the δ-rules:

$$proj_i\ (Pair\ v_1\ v_2) \longrightarrow_\delta v_i$$

The reduction $proj_1\ (Pair\ v\ (\lambda x.\,Pair\ Pair)) \longrightarrow_\delta v$ is correct, even though the right component of the pair is ill-typed, hence δ-reduction is larger than the type-erasure of δ-reduction on explicitly typed terms. Still, it contains it (direct simulation); and it does not contains more (inverse simulation) when we restrict to well-typed expressions. Both cases are really easy:

Proof: Let M be of the form $proj_i\ \tau_1\ \tau_2\ V_0$ such that $\Gamma \vdash M : \tau$. By inversion of typing rules, $\Gamma \vdash V_0 : \tau_1 \times \tau_2$. By the classification lemma, V_0 is of the form $Pair\ \tau_1\ \tau_2\ V_1\ V_2$. Observe that M reduces to V_i (**1**); and $\lceil M \rceil$ is $proj_i\ (pair\ \lceil V_1 \rceil\ \lceil V_2 \rceil)$ which reduces to $\lceil V_i \rceil$ (**2**).

Case direct: Assume $M \longrightarrow_\delta M'$. Then, M' is V_i and by (2), $\lceil M \rceil \longrightarrow_\delta \lceil V_i \rceil$.

Case inverse: Assume that $\lceil M \rceil \longrightarrow_\delta a$. Since reduction is deterministic in the untyped calculus a must be $\lceil V_i \rceil$. Hence, we may take V_i for M'.

■

Proof of Lemma 21

Assume $\Gamma \vdash M : \tau$ and $M \longrightarrow M'$. We reason by induction on the proof of reduction.

Case $(\Lambda(\alpha)\ M_0)\ \tau \longrightarrow_\iota M_0[\tau/\alpha]$: Observe that both M and M' erases to $\lceil M_0 \rceil$.

Case $(\lambda x.\, M_1)\ M_2 \longrightarrow_\beta M_1[M_2/x]$: Then M erases to $(\lambda x.\lceil M_1 \rceil)\ \lceil M_1 \rceil$ which reduces to $\lceil M_1 \rceil[\lceil M_2 \rceil/x]$ which is the erasure of M'

Case $\mathsf{proj}_i\ \tau_1\ \tau_2\ (V_1, V_2) \longrightarrow_\delta V_i$: The conclusion follows by assumption on δ-rules.

Case M is $E[N]$ and M' is $E[N']$ and $N \longrightarrow_z N'$: By induction hypothesis, we know that a certain relation (equality when z is ι or $\longrightarrow_z$ otherwise) holds between $\lceil M \rceil$ and $\lceil N \rceil$. By rule congruence for ι and rule Context otherwise, the same relation holds between $\lceil E[M] \rceil$ and $\lceil E \rceil[\lceil N \rceil]$, *i.e.* between $\lceil M \rceil$ and $\lceil N \rceil$.

Proof of Lemma 25

Case v is $\lambda x.\, a_1$: By inversion of type erasure, M is of the form $\mathcal{R}[\lambda x{:}\tau.\, M_1]$ where $\lceil M_1 \rceil$ is a_1. Since $\mathcal{R}$ is ι-normal, it is of the form $\Lambda\overline{\alpha}.[]\ \overline{\tau}$. since $\lambda x{:}\tau.\, M_1$ is an arrow type, $\overline{\tau}$ must be empty.

Case v is a partial application $c\ v_1\ \ldots v_n$: We show that then V is $\mathcal{R}[c\ \vec{\tau}\ V_1\ \ldots V_n]$ with $\lceil V_i \rceil = v_i$ by induction on n. If n is zero, then by inversion of type erasure, M is of the form $\mathcal{R}[c]$ as expected. Otherwise, by inversion of type erasure, M is an application $\mathcal{R}_n[M_1\ M_2]$ where $\lceil M_1 \rceil$ is the partial application $c\ v_1\ \ldots v_{n-1}$ and $\lceil M_2 \rceil$ is v_n. By induction hypothesis M_1 is $\mathcal{R}_1[c\ \vec{\tau}\ V_1\ \ldots V_{n-1}]$ with $\lceil V_i \rceil = v_i$. Since $\mathcal{R}_1$ is in an evaluation context, it is ι-normal, hence of the form $\Lambda\bar{\alpha}_1.[]\ \bar{\tau}_1$. From the arity of c, the type of M_1 is an arrow type. Thus $\bar{\tau}_1$ must be empty. Since $\mathcal{R}$ is applied to M_2 it cannot be a type abstraction either. Hence, $\mathcal{R}_1$ is empty. Moreover, by induction hypothesis M_2 is a value V_n. Hence M is $\mathcal{R}_n[c\ \vec{\tau}\ V_1\ \ldots V_{n-1}\ V_n]$, as expected.

Proof of Corollary 26

By Lemma 24, M is of the form $\mathcal{R}[M_0\ M_2]$ where $\lceil M_0 \rceil$ is a value v, which is either $\lambda x.\, a_1$ or the partial application $c\ v_1\ \ldots v_{n-1}$ and $\lceil M_2 \rceil$ is v. Since $\mathcal{R}$ is an evaluation context, $M_0\ M_2$ is in ι-normal form. Since $[]\ M_2$ is an evaluation context, M_0 is in ι-normal form. By Lemma 25, M_0 a value V_0. Since $V_0\ []$ is an evaluation context, M_2 is in ι-normal form. By 25, it must be a value V_0.

Moreover, by Lemma 25, V_0 is either

- $\Lambda\overline{\alpha}.\lambda x{:}\tau.\, M_1$. Since V_0 is in application position $\overline{\alpha}$ must actually be empty. Then M is of the form $\mathcal{R}[(\lambda x{:}\tau.\, M_1)\ V]$, as expected.

- $\mathcal{R}_0[c\ \vec{\tau}\ V_1\ \ldots V_{n-1}]$. Since V_0 is in an evaluation $\mathcal{R}_0$ is ι-normal, thus of the form $\Lambda\vec{\alpha}.[]\ \bar{\tau}_0$. Since V_0 in application position $\vec{\alpha}$ must be empty. From the arity of d, the application is partial and has an arrow type, hence $\bar{\tau}_0$ must be empty. Then, taking V for V_n, the term M is of the form $\mathcal{R}[c\,\vec{\tau}\,V_1\ \ldots V_n]$, as expected.

Solution of Exercise 34

Take $(\lambda x\!:\!\tau_0.\,\Lambda\alpha.\lambda\alpha\!:\!y.\,y\ )\ M_0$ for M where $\Gamma \vdash M_0 : \tau_0$. We have $\Gamma \vdash_{\mathsf{F}} M : \forall\alpha.\,\alpha \to \alpha$ where M is syntactically in ML, but cannot be typed in ML because function bodies cannot have polymorphic types. ∎

Bibliography

▷ A tour of scala: Implicit parameters. Part of scala documentation.

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, March 1996.

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2–3):81–116, December 1995.

▷ Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *ACM International Conference on Functional Programming (ICFP)*, pages 157–168, September 2008.

▷ Lennart Augustsson. Implementing Haskell overloading. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 65–73, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X.

▷ Nick Benton and Andrew Kennedy. Exceptional syntax journal of functional programming. *J. Funct. Program.*, 11(4):395–410, 2001.

▷ Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. *Testing Polymorphic Properties*, pages 125–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-11957-6. doi: 10.1007/978-$\sigma_2$3-$\sigma_2$642-$\sigma_2$11957-$\sigma_2$6_8.

▷ Richard Bird and Lambert Meertens. Nested datatypes. In *International Conference on Mathematics of Program Construction (MPC)*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

Nikolaj Skallerud Bjørner. Minimal typing derivations. In *In ACM SIGPLAN Workshop on ML and its Applications*, pages 120–126, 1994.

Daniel Bonniot. *Typage modulaire des multi-méthodes*. PhD thesis, École des Mines de Paris, November 2005.

▷ Daniel Bonniot. Type-checking multi-methods in ML (a modular approach). In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 2002.

▷ Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33:309–338, 1998.

▷ Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

▷ Yufei Cai, Paolo G. Giarrusso, and Klaus Ostermann. System F-omega with equirecursive types for datatype-generic programming. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 30–43. ACM, 2016. doi: 10.1145/2837614.2837660.

Luca Cardelli. An implementation of f¡:. Technical report, DEC Systems Research Center, 1993.

Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.

▷ Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. The MLton compiler, 2007.

▷ Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

▷ Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 38–49, January 2005.

▷ Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 54–65, June 2007.

▷ Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

Julien Crétin and Didier Rémy. Extending System F with Abstraction over Erasable Coercions. In *Proceedings of the 39th ACM Conference on Principles of Programming Languages*, January 2012.

▷ Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–70, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4.

Joshua Dunfield. Greedy bidirectional polymorphism. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 15–26, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-509-3. doi: http://doi.acm.org/10.1145/1596627.1596631.

▷ Ken-etsu Fujita and Aleksy Schubert. Existential type systems with no types in terms. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, pages 112–126, 2009. doi: 10.1007/978-$\sigma_2$3-$\sigma_2$642-$\sigma_2$02273-$\sigma_2$9_10.

Jun Furuse. Extensional polymorphism by flow graph dispatching. In Ohori (2003), pages 376–393. ISBN 3-540-20536-5.

▷ Jun Furuse. Extensional polymorphism by flow graph dispatching. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science*. Springer, November 2003b.

▷ Jacques Garrigue. Relaxing the value restriction. In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, April 2004.

▷ Nadji Gauthier and François Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 150–161, September 2004. doi: http://doi.acm.org/10.1145/1016850.1016872.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université Paris 7, June 1972.

▷ Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1990.

▷ Dan Grossman. Quantified types in an imperative language. *ACM Transactions on Programming Languages and Systems*, 28(3):429–475, May 2006.

▷ Bob Harper and Mark Lillibridge. ML with callcc is unsound. Message to the TYPES mailing list, July 1991.

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–345. MIT Press, 2005.

▷ Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

▷ J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus.* Cambridge University Press, 1986.

▷ Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *ACM SIGPLAN Conference on History of Programming Languages*, June 2007.

Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris 7, September 1976.

▷ John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

▷ Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 160–169, New York, NY, USA, 1995a. ACM. ISBN 0-89791-719-7.

Mark P. Jones. Typing Haskell in Haskell. In *In Haskell Workshop*, 1999a.

Mark P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995b. ISBN 0-521-47253-9.

▷ Mark P. Jones. Typing Haskell in Haskell. In *Haskell workshop*, October 1999b.

▷ Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, 1997.

▷ Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(01):1, 2006.

Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 193–204, New York, NY, USA, 1992. ACM. ISBN 0-89791-481-3. doi: http://doi.acm.org/10.1145/141471.141540.

▷ Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, pages 206–220. Springer, May 1990.

▷ Oleg Kiselyov. Higher-kinded bounded polymorphism. web page.

▷ Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Communications of the ACM*, 8(2):89–101, 1965.

▷ Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

▷ Didier Le Botlan and Didier Rémy. Recasting MLF. *Information and Computation*, 207(6): 726–785, 2009. ISSN 0890-5401. doi: 10.1016/j.ic.2008.12.006.

▷ Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris 7, June 1992.

▷ Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54, January 2006.

▷ Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/349214.349230.

▷ John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–57, January 1988.

▷ Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 382–401, 1990.

▷ David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer, June 2003.

Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 279–303, London, UK, 1999. Springer-Verlag. ISBN 3-540-66156-5.

▷ Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

▷ Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–283, January 1996.

▷ John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2–3):211–249, 1988.

▷ John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

▷ Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 63–74, January 2009.

J. Garrett Morris and Mark P. Jones. Instance chains: type class programming without overlapping instances. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 375–386, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: http://doi.acm.org/10.1145/1863543.1863596.

▷ Greg Morrisett and Robert Harper. Typed closure conversion for recursively-defined functions (extended abstract). In *International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

▷ Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

▷ Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, April 1984.

▷ Hiroshi Nakano. A modality for recursion. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 255–266, June 2000.

▷ Hiroshi Nakano. Fixed-point logic with the approximation modality and its Kripke completeness. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, volume 2215 of *Lecture Notes in Computer Science*, pages 165–182. Springer, October 2001.

▷ Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. Functional logic overloading. pages 233–244, 2002. doi: http://doi.acm.org/10.1145/565816.503294.

▷ Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 135–146, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7.

▷ Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

▷ Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 41–53, 2001.

Atsushi Ohori, editor. *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings*, volume 2895 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-20536-5.

▷ Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

▷ Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: a new foundation for generic programming. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 35–44, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254070.

▷ Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Online lecture notes, January 2009.

▷ Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Manuscript, April 2004.

▷ Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, January 1993.

Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: http://doi.acm.org/10.1145/62678.62697.

▷ Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

▷ Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

▷ Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

▷ François Pottier. Notes du cours de DEA "Typage et Programmation", December 2002.

François Pottier. A typed store-passing translation for general references. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, Austin, Texas, January 2011. Supplementary material.

François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of Functional Programming*, 23(1):38–144, January 2013.

François Pottier. Hindley-Milner elaboration in applicative style. In *Proceedings of the 2014 ACM SIGPLAN International Conference on Functional Programming (ICFP'14)*, September 2014.

▷ François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. Submitted for publication, October 2012.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 173–184, September 2013.

▷ François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

▷ François Pottier and Didier Rémy. The essence of ML type inference. Draft of an extended version. Unpublished, September 2003.

▷ Didier Rémy. Simple, partial type-inference for System F based on type-containment. In *Proceedings of the tenth International Conference on Functional Programming*, September 2005.

▷ Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346. Springer, April 1994a.

▷ Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming: Types, Semantics and Language Design*. MIT Press, 1994b.

▷ Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

Didier Rémy and Boris Yakobowski. Efficient Type Inference for the MLF language: a graphical and constraints-based approach. In *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 63–74, Victoria, BC, Canada, September 2008. doi: http://doi.acm.org/10.1145/1411203.1411216.

▷ John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, April 1974.

▷ John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

▷ John C. Reynolds. Three approaches to type structure. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer, March 1985.

François Rouaix. Safe run-time overloading. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 355–366, 1990. doi: http://doi.acm.org/10.1145/96709.96746.

▷ Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. Cochis: Deterministic and coherent implicits. Technical report, KU Leuven, May 2017.

▷ Christian Skalka and François Pottier. Syntactic type soundness for HM(X). In *Workshop on Types in Programming (TIP)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, July 2002.

Lau Skorstengaard. An Introduction to Logical Relations. *arXiv e-prints*, art. arXiv:1907.11133, July 2019.

Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. In *Science of Computer Programming*, 1994.

Morten Heine Sørensen and Pawel Urzyczyn. *Studies in Logic and the Foundations of Mathematics*, chapter Lectures on the Curry-Howard Isomorphism. Elselvir Science Inc, 2006.

▷ Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In Sofiène Tahar, Otmame Ait-Mohamed, and César Muñoz, editors, *TPHOLs 2008: Theorem Proving in Higher Order Logics, 21th International Conference*, Lecture Notes in Computer Science. Springer, August 2008.

▷ Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.

▷ Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000.

▷ Peter J. Stuckey and Martin Sulzmann. A theory of overloading. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 167–178, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8.

▷ W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):pp. 198–212, 1967. ISSN 00224812.

▷ Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 11(2):245–296, 1994.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

▷ Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179(1):1–18, 2002.

▷ Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, September 2004.

▷ Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

▷ Philip Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, September 1989.

▷ Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1–3):201–226, May 2007.

▷ Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–76, January 1989.

Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

▷ J. B. Wells. The essence of principal typings. In *International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer, 2002.

▷ J. B. Wells. The undecidability of Mitchell's subtyping relation. Technical Report 95-019, Computer Science Department, Boston University, December 1995.

▷ J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

▷ Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8 (4):343–356, December 1995.

▷ Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

▷ Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 119–135, Cham, 2014. Springer International Publishing. ISBN 978-3-319-07151-0.